DETERMINISTIC CLOCK GATING FOR LOW POWER VLSI DESIGN

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Technology

in

VLSI Design and Embedded System

By

SURESH KUMAR AANANDAM

Roll No : 20507007

Department of Electronics & Communication Engineering

National Institute of Technology, Rourkela

Rourkela, orissa – 769008

2007

DETERMINISTIC CLOCK GATING FOR LOW POWER VLSI DESIGN

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Technology

in

VLSI Design and Embedded System

By

SURESH KUMAR AANANDAM

Roll No : 20507007

Under the Guidance of

Prof. K.K.MAHAPATRA

Department of Electronics & Communication Engineering

National Institute of Technology, Rourkela

Rourkela, orissa – 769008

2007

NATIONAL INSTITUTE OF TECHNOLOGY
ROURKELA

CERTIFICATE

This is to certify that the thesis titled, " **Deterministic Clock Gating for Low Power VLSI Design "** submitted by Mr. **Suresh Kumar Aanandam** (Roll No : 20507007), in partial fulfillment for the award of Master of Technology degree in **Electronics & Communication Engineering** with the specialization in " **VLSI Design and Embedded System "** during the session 2006-2007 at the National Institute of Technology, Rourkela (Deemed University) is an authentic work carried out by him under my supervision and guidance.

To the best of my knowledge, this matter embodied in the thesis has not been submitted at any other university / institute for the award of any Degree or Diploma.

Prof. K.K. MAHAPATRA

Date :

Department of E.C.E.
National Institute of Technology
Rourkela – 769008.

ACKNOWLEDGEMENTS

I would like to extend my gratitude & my sincere thanks to my honorable, esteemed supervisor **Prof. K.K.Mahapatra,** Department of Electronics and Communication Engineering. He is not only a great lecturer with deep vision but also a kind person. His knowledge and company at the time of crisis would be remembered lifelong.

I would like to thank all my teachers **Prof. G.S. Rath**, **Prof. G. Panda, Prof. S.K. Patra** and **Dr. S. Meher** for providing a solid background for my studies and research thereafter. They have been great sources of inspiration to me and I thank them from the bottom of my heart.

I would like to thank all faculty members and staff of the Department of Electronics and Communication Engineering, N.I.T. Rourkela for their generous help in various ways for the completion of this thesis.

I am especially indebted to my parents for their love, sacrifice, and support. They are my first teachers and set great examples for me about how to live, study, and work.

Last but not least, I would like to thank all my friends, who made my stay in Rourkela an unforgettable experience.

SURESH KUMAR AANANDAM

CONTENTS

ABSTRACT

The demand for power-sensitive design has grown significantly in recent years due to tremendous growth in portable applications. Consequently, the need for power efficient design techniques has grown considerably. Several efficient design techniques have been proposed to reduce both dynamic as well as static power in state-of-the-art VLSI circuit applications. With the scaling of technology and the need for higher performance and more functionality, power dissipation is becoming a major bottleneck for microprocessor designs. Clock power is significant in high-performance processors.

Deterministic Clock Gating (DCG) technique effectively reduces the clock power. DCG is based on the key observation that for many of the pipelined stages of a modern processor, the circuit block usage in the near future is known a few cycles ahead of time. DCG exploits this advance knowledge to clock-gate the unused blocks. Because individual circuit usage varies within and across applications, not all the circuits are used all the time, giving rise to power reduction opportunity. By ANDing the clock with a gate-control signal, clock-gating essentially disables the clock to a circuit whenever the circuit is not used, avoiding power dissipation due to unnecessary charging and discharging of the unused circuits.

Results show that DCG is very effective in reducing clock power. 25 – 33 % power consumption is reduced by using this method. As high-performance processor pipelines get deeper and power becomes a more critical factor, DCG's effectiveness and simplicity will continue to be important.

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

1.1. INTRODUCTION

In recent years, the demand for power-sensitive designs has grown significantly. This tremendous demand has mainly been due to the fast growth of battery-operated portable applications such as notebook and laptop computers, personal digital assistants, cellular phones, and other portable communication devices. Semiconductor devices are aggressively scaled each technology generation to achieve high-performance and high integration density. Due to increased density of transistors in a die and higher frequencies of operation, the power consumption in a die is increasing every technology generation. Supply voltage is scaled to maintain the power consumption within limit.

However, scaling of supply voltage is limited by the high-performance requirement. Hence, the scaling of supply voltage only may not be sufficient to maintain the power density within limit, which is required for power-sensitive applications. Circuit technique and system-level techniques are also required along with supply voltage scaling to achieve low-power designs.
In the nano-meter regime, a significant portion of the total power consumption in high performance digital circuits is due to leakage currents. Because high-performance systems are constrained to a predefined power budget, the leakage power reduces the available power, impacting performance. It also contributes to the power consumption during standby operation, reducing battery life. Hence, techniques are necessary to reduce leakage power while maintaining the high performance. Moreover, as different components of leakage are becoming important with technology scaling, each leakage reduction technique needs reevaluation in scaled technologies where sub-threshold conduction is not the only leakage mechanism. New low-power circuit techniques are required to reduce total leakage in high-performance nano-scale circuits.

A spectrum of circuit techniques including transistor sizing, clock gating, multiple and dynamic supply voltage are there to reduce the dynamic power. For low-leakage design, different circuit techniques including, dual V_{th}, forward/reverse bias, dynamically varying the V_{th} during run time, sleep transistor, natural stacking are there.

1.2. POWER DISSIPATION IN VLSI CIRCUITS

The total power dissipation in a circuit conventionally consists of two components, namely, the static and dynamic power dissipation.

1.2.1. Dynamic power

For dynamic power dissipation there are two components one is switching power due to charging and discharging of load capacitance. The other is the short circuit power due to the nonzero rise and fall time of input waveforms. The switching power of a single gate can be expressed as

$$P_D = \alpha C_L V_{DD}^2 f$$

Where α is the switching activity,

f is the operation frequency,

C_L is the load capacitance,

V_{DD} is the supply voltage.

The short circuit power of an unloaded inverter can be approximately given by

$$P_{SC} = \beta (V_{DD} - V_{th})^3 \tau / 12T$$

Where β is the transistor coefficient,

τ is the rise/fall time,

T (1/f) is the delay.

1.2.2. Leakage power

There are three dominant components of leakage in a MOSFET in the nanometer regime:

(1) Sub-threshold leakage, which is the leakage current from drain to source (I_{sub}).

(2) Direct tunneling gate leakage which is due to the tunneling of electron (or hole) from the bulk silicon through the gate oxide potential barrier into the gate.

(3) The source/substrate and drain/substrate reverse-biased p-n junction leakage.

1.3. DYNAMIC POWER REDUCTION TECHNIQUES

Though the leakage power increases significantly in every generation with technology scaling, the dynamic power still continues to dominate the total power dissipation of the general purpose microprocessors. Effective circuit techniques to reduce the dynamic power consumption include transistor size and interconnect optimization, gated clock, multiple

supply voltages and dynamic control of supply voltage. Incorporating the above approaches in the design of nano-scale circuits, the dynamic power dissipation can be reduced significantly. Other techniques such as instruction set optimization, memory access reduction and low complexity algorithms are also there to reduce the dynamic power dissipation in both logics and memories.

1.3.1. Transistor sizing and interconnect optimization

The best way to reduce the junction capacitance as well as the overall gate capacitance is to optimize the transistor size for a particular performance. Sizing techniques can be mainly divided into two types.

- Path-based optimization.
- Global optimization.

In path-based optimization, gates in the critical paths are upsized to achieve the desired performance, while the gates in the off critical paths are down sized to reduce power consumption.

In global optimization, all gates in a circuit are globally optimized for a given delay.

1.3.2. Clock gating

Clock gating is an effective way of reducing the dynamic power dissipation in digital circuits. In a typical synchronous circuit such as the general purpose microprocessor, only a portion of the circuit is active at any given time. Hence, by shutting down the idle portion of the circuit, the unnecessary power consumption can be prevented. One of the ways to achieve this is by masking the clock that goes to the idle portion of the circuit.

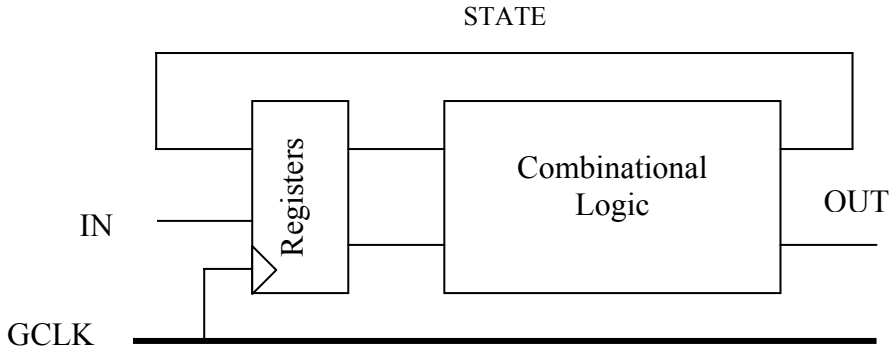

Fig 1.1. Single clock, flip-flop-based FSM.

This prevents unnecessary switching of the inputs to the idle circuit block, reducing the dynamic power. The input to the combinational logic comes through the registers, which are usually composed of sequential elements, such as D flip-flops (Fig. 1.1.).

A gated clock design can be obtained by modifying the clocking structure shown in Fig.1.1. A control signal (f_a) is used to selectively stop the local clock (LCLK) when the combinational block is not used. The local clock is blocked when f_a is high. The latch shown in Fig.1.2 is necessary to prevent any glitches in f_a from propagating to the AND gate when the global clock (GCLK) is high. The circuit operates as follows.

The signal f_a is only valid before the rising edge of the global clock. When the global clock is low, the latch is transparent, however, f_a does not affect the AND gate. If f_a is high during the low-to-high transition of the global clock, then the global clock will be blocked by the AND gate and local clock will remain at low. Power saving using gated clock technique strongly depends on the efficient synthesis and optimization of dedicated clock-stopping circuitry.

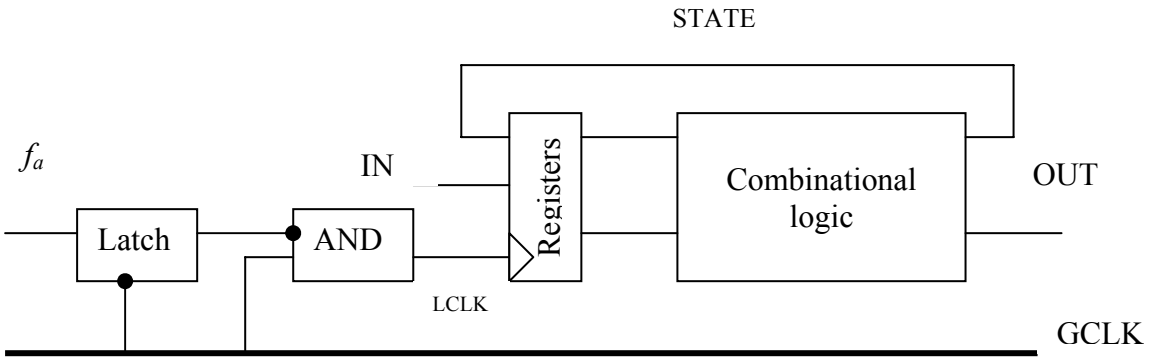

Fig.1.2. Schematic diagram of gated clock design

Effective clock gating requires a methodology that determines which circuits are gated, when, and for how long. Clock-gating schemes that either result in frequent toggling of the clock-gated circuit between enabled and disabled states, or apply clock gating to such small blocks that the clock-gating control circuitry is almost as large as the blocks themselves, incur large overhead. This overhead may result in power dissipation to be higher than that without clock gating.

1.3.3. Low-voltage operation

Supply voltage scaling was originally developed for switching power reduction. It is an effective method for switching power reduction because of the quadratic dependency of switching power on supply voltage. However, since the gate delay increases with decreasing V_{DD}, globally lowering V_{DD} degrades the overall circuit performance. To achieve low-power benefits without compromising performance, two ways of lowering supply voltage can be employed: static and dynamic supply scaling.

In Static supply voltage scaling schemes, higher supply voltage is used in the critical paths of the circuit, while lower supply voltages are used in the off critical paths.
In Dynamic supply voltage scaling schemes, the highest supply voltage delivers the highest performance at the fastest designed frequency of operation. When performance demand is low, supply voltage and clock frequency is lowered, just delivering the required performance with substantial power reduction.

1.4. LEAKAGE POWER REDUCTION TECHNIQUES

The techniques to reduce leakage energy utilizing the slack without impacting performance can be categorized based on when and how they utilize the available timing slack e.g. dual V_{th} statically assigns high V_{th} to some transistors in non-critical paths at the design time so as to reduce leakage current. The techniques, which utilize the slack in run time, can be divided into two groups depending on whether they reduce standby leakage or active leakage. Standby leakage reduction techniques put the entire system in a low leakage mode when computation is not required. Active leakage reduction techniques slow down the system by dynamically changing the V_{th} to reduce leakage when maximum performance is not needed.

1.4.1. Design time techniques

Design time techniques exploit the delay slack in non-critical paths to reduce leakage. These techniques are static; once it is fixed, it cannot be changed dynamically while the circuit is operating.

Dual threshold CMOS logic:

In this logic, a high V_{th} can be assigned to some transistors in the non-critical paths so as to reduce sub-threshold leakage current, while the performance is not sacrificed by using low V_{th}

transistors in the critical path(s). No additional circuitry is required, and both high performance and low leakage can be achieved simultaneously.

Different Dual threshold CMOS techniques are

- Changing doping profile.
- Higher oxide thickness
- Large channel length

1.4.2. Run time techniques

Standby leakage reduction techniques place certain sections of the circuitry in standby mode (low leakage mode) when they are not required.

Different Standby leakage reduction techniques are

- Natural transistor stacks.
- Sleep transistor (forced stacking).
- Forward/reverse body biasing.

Active leakage reduction techniques are intermittently slows down the faster circuitry and reduces the leakage power consumption as well as the dynamic power consumption when maximum performance is not required.

Dynamic V_{th} scaling (DVTS) scheme uses body biasing to adaptively change V_{th} based on the performance demand. The lowest V_{th} is delivered, if the highest performance is required. When performance demand is low, clock frequency is lowered and V_{th} is raised to reduce the run-time leakage power dissipation. In cases when there is no workload at all, the V_{th} can be increased to its upper limit to significantly reduce the standby leakage power.

1.4.3. Cache memories

Circuit techniques to reduce leakage in cache memories are

- Source biasing scheme.
- Forward/reverse body biasing scheme.
- Dynamic VDD scheme
- Leakage biased scheme
- Negative word line scheme

Chapter 2

CLOCK GATING

2.1. INTRODUCTION

Clock power is a major component of microprocessor power mainly because the clock is fed to most of the circuit blocks in the processor, and the clock switches every cycle. Thus the total clock power is a substantial component of total microprocessor power dissipation.

Clock-gating is a well-known technique to reduce clock power. Because individual circuit usage varies within and across applications, not all the circuits are used all the time, giving rise to power reduction opportunity. By ANDing the clock with a gate-control signal, clock-gating essentially disables the clock to a circuit whenever the circuit is not used, avoiding power dissipation due to unnecessary charging and discharging of the unused circuits. Specifically, clock-gating targets the clock power consumed in pipeline latches and dynamic-CMOS-logic circuits (e.g., integer units, floating-point units, and word-line decoders of caches) used for speed and area advantages over static logic.

Effective clock-gating, however, requires a methodology that determines which circuits are gated, when, and for how long. Clock-gating schemes that either result in frequent toggling of the clock-gated circuit between enabled and disabled states, or apply clock-gating to such small blocks that the clock-gating control circuitry is almost as large as the blocks themselves, incur large overhead. This overhead may result in power dissipation to be higher than that without clock-gating.

2.1. PRINCIPLE OF CLOCK-GATING

The clock network in a microprocessor feeds clock to sequential elements like flip-flops and latches, and to dynamic logic gates, which are used in high-performance execution units and array address decoders (e.g. *D*-cache word-line decoder). At a high level, gating the clock to a latch or a logic gate by ANDing the clock with a control signal prevents the unnecessary charging/discharging of the capacitances when the circuit is idle, and saves the circuit’s clock power.

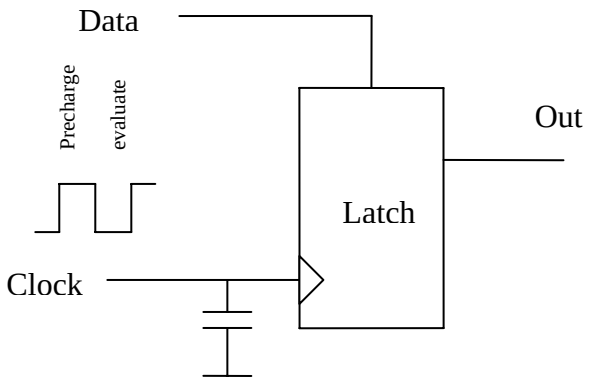

Fig 2.1 A latch element

Fig. 2.1 shows the schematic of a latch element. C_g is the latch's cumulative gate capacitance connected to the clock. Because the clock switches every cycle, C_g charges and discharges every cycle and consumes significant amount of power. Even if the inputs do not change from one clock to the next, the latch still consumes clock power.

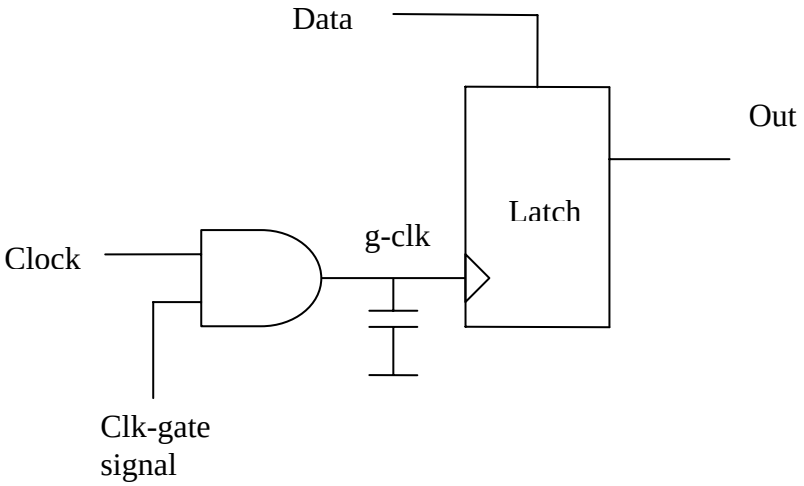

Fig 2.2 clock gating a latch element

In Fig. 2.2, the clock is gated by ANDing it with a control signal, which we refer as *Clk-gate signal*. When the latch is not required to switch state, *Clk-gate signal* is turned off and the clock is not allowed to charge/discharge C_g, saving clock power. Because the latches of an operand (32 or 64 b) can be driven by an AND gate, the capacitance of the AND gate itself is much smaller than the sum of multiple C_g of these latches. Hence, we can get a net power saving.

Now, let us consider a dynamic logic cell, the schematic of which is shown in Fig. 2.3.

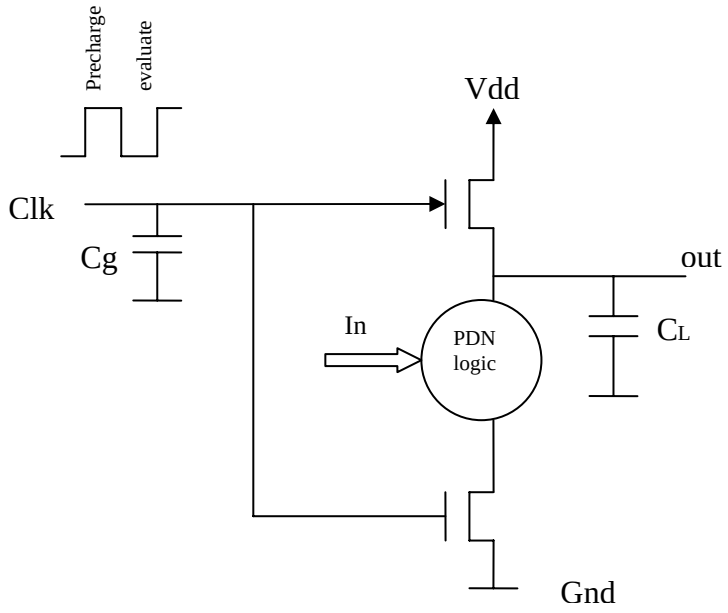

Fig 2.3 A dynamic logic gate

C_g is the effective gate capacitance that appears as a capacitive load to the clock, and C_L is the capacitive load to the dynamic logic cell. Similar to the latch, the dynamic logic's C_g also charges and discharges every cycle and consumes power.

In addition to C_g, C_L also consumes power: at the *precharge* phase of the clock, C_L charges through the PMOS *precharge* transistor and during the *evaluate* phase, it discharges or retains value depending on the input to the pull-down logic. Whether C_L consumes power or not depends on *both* the current input and previous output. There are two cases:

(1) If C_L holds logic "1" at the end of a cycle, and the next cycle output evaluates to a "1", then C_L does not consume any power: Precharging an already-charged C_L does not consume power unless there are leakage losses. Because the next output is a "1", there is no discharging.

(2) If C_L holds a "0" at the end of a cycle, C_L consumes precharge power, irrespective of what the inputs are in the next cycle. Even if the input does not change, this precharge power is consumed. If the next output is a "1", no discharging occurs; otherwise, more power is consumed in discharging C_L.

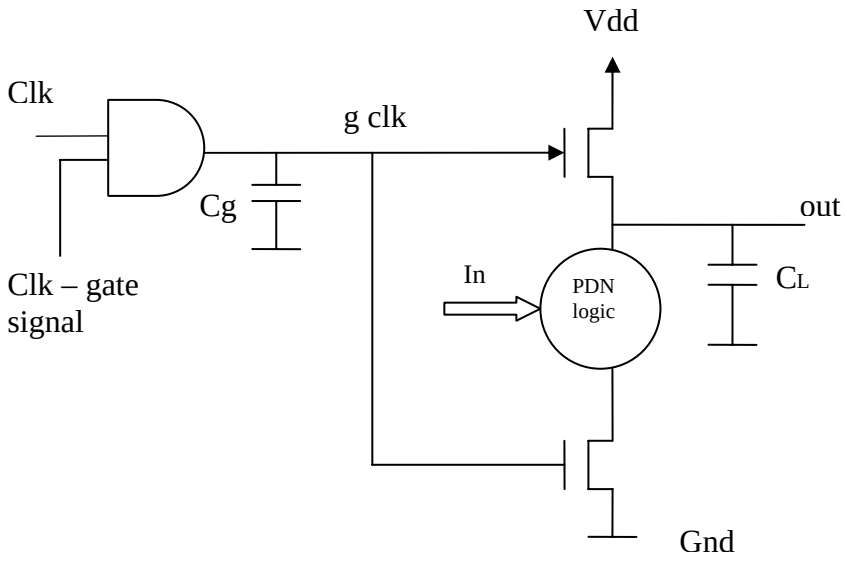

Fig 2.4 clock gating a dynamic logic gate

Fig. 2.4 shows the same dynamic logic cell with gated clock. If the dynamic logic cell is not used in a cycle, *Clk-gate signal* prevents both C_g and C_L from switching in the cycle. While clock-gating latches reduce only unnecessary clock power due to C_g, clock-gating dynamic logic reduces unnecessary dissipation of not only the clock power due to C_g, but also the dynamic logic power due to C_L. Here also, because the AND gate's capacitance itself is much smaller than $C_g + C_L$, there is a net power saving. Moreover, a single AND gate can be used to gate the clock to a large number of dynamic logic cells.

The concept of circuit-level clock-gating can be achieved by two good architectural methodologies. They are Pipeline balancing (PLB) and Deterministic clock-gating.
Pipeline balancing method (PLB) exploits the inherent variation of instruction level parallelism (ILP) within a program. PLB uses heuristics to predict a program's instruction level parallelism (ILP). If the degree of ILP in the next window is predicted to be lower than the width of the pipeline, PLB clock-gates a cluster of pipeline components during the window.

In contrast to PLB's predictive methodology, *Deterministic clock-gating (DCG)* is based on the key observation that for many of the pipeline stages in a modern processor, a circuit block usage in a specific cycle in the near future is *deterministically* known a few cycles ahead of time. DCG exploits this advance knowledge to clock-gate the unused blocks.

Chapter 3

GENERAL PURPOSE PROCESSOR

3.1. INTRODUCTION

In designing a CPU, we must first define its instruction set and how the instructions are encoded and executed. We need to answer questions such as how many instructions do we want? What are the instructions? What operation code (opcode) do we assign to each of the instructions? How many bits do we use to encode an instruction?

Once we have decided on the instruction set, we can proceed to designing a datapath that can execute all the instructions in the instruction set. In this step we are creating a custom datapath, so we need to answer questions such as what functional units do we need? How many registers do we need? Do we use a single register file or separate registers? How the different units are connected together?

Finally, we can design the control unit. Just like the dedicated microprocessor, the control unit asserts the control signals to the datapath. This finite-state machine cycles through three main steps or states: 1) fetch an instruction; 2) decode the instruction; and 3) execute the instruction. The control unit performs these steps by sending the appropriate control signals to the datapath or to external devices.

Instructions in your program are usually stored in external memory, so in addition to the CPU, there is external memory that is connected to the CPU via an address bus and a data bus. Hence, step 1 (fetch an instruction) usually involves the control unit setting up a memory address on the address bus and telling the external memory to output the instruction from that memory location onto the data bus. The control unit then reads the instruction from the data bus. To keep our design simple, instead of having external memory, we will put the memory directly inside the CPU and implemented simply as a 64-byte array. In fact, there are real CPUs with internal program memory.

For step 2 (decode the instruction) the control unit extracts the opcode bits from the instruction and determines what the current instruction is by jumping to the state that has been assigned for executing that instruction. Once in that particular state, the finite-state machine performs step 3 by simply asserting the appropriate control signals for controlling the datapath to execute that instruction.

3.2. GENERAL PURPOSE MICROPROCESSOR

The instructions that our general-purpose microprocessor can execute and the corresponding encoding are defined in Figure 1. The Instruction column shows the syntax and mnemonic to use for the instruction when writing a program in assembly language. The Encoding column shows the binary encoding for the instructions and the Operation column shows the actual operation of the instruction. The instructions are separated into four categories:

1) Data movement instructions for transferring data between the accumulator, the general registers and the memory.
2) Jump instructions for changing the instruction execution sequence.
3) Arithmetic and logical instructions for performing arithmetic and logics. and
4) Input / Output and miscellaneous instructions. There are five data movement instructions, eight jump instructions, ten arithmetic and logic instructions, two input/output instructions, and two miscellaneous instructions.

The number of instructions implemented determines the number of bits required to encode all the instructions. All instructions are encoded using one byte except for instructions that have a memory address as one of its operand, in which case a second byte for the address is needed. The encoding scheme uses the first four bits as the opcode. Depending on the opcode, the last four bits are interpreted differently as follows.

3.2.1. Two Operand Instructions

If the instruction requires two operands, it always uses the accumulator (A) for one operand. If the second operand is a register then the last three bits in the encoding specifies the register file number. An example of this is the LDA (load accumulator from register) instruction where it loads the accumulator with the content of the register file number specified in the last three bits of the encoding. Another example is the ADD (add) instruction where it adds the content of the accumulator with the content of the specified register file and put the result in the accumulator. The result of all arithmetic and logical operations is stored in the accumulator.

The LDI (load accumulator with immediate value) is also a two-operand instruction. However, the second operand is an immediate value that is obtained from the second byte of

the instruction itself (iiiiiiii). These eight bits are interpreted as a signed number and is loaded into the accumulator.

3.1.2. One Operand Instructions

One-operand instructions always use the accumulator and the result is stored back in the accumulator. In this case, the last four bits in the encoding are used to further decode the instruction. An example of this is the INC (increment accumulator) instruction. The opcode (1110) is used by all the one-operand arithmetic and logical instructions. The last four bits (0001) specify the INC instruction.

3.2.3. Instructions Using a Memory Address

For instructions that have a memory address as one of its operand, an additional six bits are needed in order to access the 64 bytes of memory space. These six bits (aaaaaa) are specified in the six least significant bits of the second byte of the instruction. An example is the LDM (load accumulator from memory) instruction. The address of the memory location where the data is to be loaded from is specified in the second byte. In this case, the last four bits of the first byte and the first two bits in the second byte are not used and are always set to 0. All the absolute jump instructions follow this format.

3.2.4. Jump Instructions

For jump instructions, the last four bits of the encoding also serves to differentiate between absolute and relative jumps. If the last four bits are zeros, then it is an absolute jump, otherwise, they represent a sign and magnitude format relative displacement from the current location as specified in the program counter (PC). For example, the two-byte encoding 0110 0000 0000 0100 specifies an absolute unconditional jump to memory location 4. The first four bits (0110) specify the unconditional jump. The second four bits (0000) specify an absolute jump. The last six bits (000100) specify the memory address.

On the other hand, the one-byte encoding 0110 0100 specifies a relative unconditional jump to PC + 4. Again, the first four bits (0110) specify the unconditional jump. The next four bits (0100) specify that it is a relative jump because it is not zero. The relative position to jump to is +4 because the first bit is a 0, which is for forward and the last three bits evaluate to 4. To jump backward by four locations, we would use 1100 instead.

Two conditional flags (zero and positive) are used for conditional jumps. These flags are set or reset depending on the value of the accumulator when the accumulator is written to. Instructions that modify the accumulator include LDA, LDM, LDI, all the arithmetic and logic instructions, and IN. For example, if the result of the ADD instruction is a positive number, then the zero flag will be reset and the positive flag will be set. A conditional jump then reads the value of these flags to see whether to jump or not. The JZ instruction will not jump after the previous ADD instruction, where as the JP instruction will perform the jump.

Notations:

A = accumulator.

R = general register.

M = memory.

rrr = three bits for specifying the general register number (0 – 7).

aaaaaa = six bits for specifying the memory address.

iiiiiiii = an eight bit signed number.

PC = program counter.

smmm = four bits for specifying the relative jump displacement in sign and magnitude format. The most significant bit (s) determines whether to jump forward or backward (0 = forward, 1 = backward). The last three bits (mmm) specify the number of locations to increment or decrement from the current PC location.

3.2.5. Instruction set

Instruction	Encoding	Operation	Comment
LDA A,rrr	0001 0rrr	A ← R[rrr]	Load accumulator from register
STA rrr,A	0010 0rrr	R[rrr] ← A	Load register from accumulator
LDM A,aaaaaa	0011 0000 00 aaaaaa	A ← M[aaaaaa]	Load accumulator from memory
STM aaaaaa,A	0100 0000 00 aaaaaa	M[aaaaaa] ← A	Load memory from accumulator
LDI A,iiiiiiii	0101 0000 iiiiiiii	A ← iiiiiiii	Load accumulator with immediate value(iiiiiiii is a signed number)

Table 3.1. Data movement instructions

Instruction	Encoding	Operation	Comment
JMP absolute	0110 0000 00 aaaaaa	PC = aaaaaa	Absolute unconditional jump

JMPR relative	0110 smmm	if (smmm!= 0) then
		if (s == 0) then PC = PC+mmm;
		else PC = PC-mmm;
			Relative unconditional jump

JZ absolute	0111 0000 00 aaaaaa	if (A == 0) then PC = aaaaaa	Absolute jump if A is zero
JZR relative	0111 smmm	if (A == 0 and smmm!= 0) then if (s == 0) then PC = PC+mmm; else PC = PC-mmm;	Relative jump if A is zero
JNZ absolute	1000 0000 00 aaaaaa	if (A != 0) then PC = aaaaaa	Absolute jump if A is notzero
JNZR relative	1000 smmm	if (A != 0 and smmm!= 0) then if (s == 0) then PC = PC+mmm; else PC = PC-mmm;	Relative ump if A is notzero
JP absolute	1001 0000 00 aaaaaa	if (A == possitive) then PC = aaaaaa	Absolute jump if A is possitive
JPR relative	1001 smmm	if (A == positive and smmm!= 0) then if (s == 0) then PC = PC+mmm; else PC = PC-mmm;	Relative ump if A is possitive

Table 3.2. Jump instructions

Instruction	Encoding	Operation	Comment
AND A,rrr	1010 0rrr	A ← A AND R[rrr]	Accumulator AND register
OR A,rrr	1011 0rrr	A ← A OR R[rrr]	Accumulator OR register
ADD A,rrr	1100 0rrr	A ← A + R[rrr]	Accumulator + register
SUB A,rrr	1101 0rrr	A ← A - R[rrr]	Accumulator - register
NOT A	1110 0000	A ← NOT A	Invert accumulator
INC A	1110 0001	A ← A + 1	Increment accumulator
DEC A	1110 0010	A ← A - 1	Decrement accumulator
SHFL A	1110 0011	A ← A << 1	Shift accumulator let
SHFR A	1110 0100	A ← A >> 1	Shift accumulator right
ROTR A	1110 0101	A ← Rotate right (A)	Rotate accumulator right

Table 3.3. Arithmetic and logical instructions

Instruction	Encoding	Operation	Comment
In A	1111 0000	A ← input	Input to accumulator
Out A	1111 0001	Output ← A	Output from accumulator
HALT	1111 0010	Halt	Halt execution
NOP	0000 0000	No operation	No operation

Table 3.4. Input / Output and Miscellaneous instructions

3.3. DATAPATH

Having defined the instruction set for our general microprocessor, we are now ready to design the custom datapath that can execute all the operations as defined by all the instructions. The resulting datapath is shown in Fig. 3.1.

The width of the datapath is eight bits, i.e. all the connections for data movement are eight bits wide (thicker lines). The remaining thinner control lines are all one bit wide unless the name for that control line has a number subscript such as faddr_$dp_{2,1,0}$, in which case there are as many lines as the subscript numbers. For example, the control line label *rfaddr_*$dp_{2,1,0}$ is actually composed of three separate lines.

3.3.1. Input multiplexer

The 4-to-1 input mux at the top of the datapath drawing selects one of four different inputs to be written into the accumulator. These four inputs, starting from the left, are:

(1) *imm_dp* for getting the immediate value from the LDI instruction and storing it into the accumulator.

(2) *input_dp* for getting a user input value for the IN instruction;

(3) The next input selection allows the content of the register file to be written to the accumulator as used by the LDA instruction.

(4) Allows the result of the ALU and the shifter to be written to the accumulator as used by all the arithmetic and logical instructions.

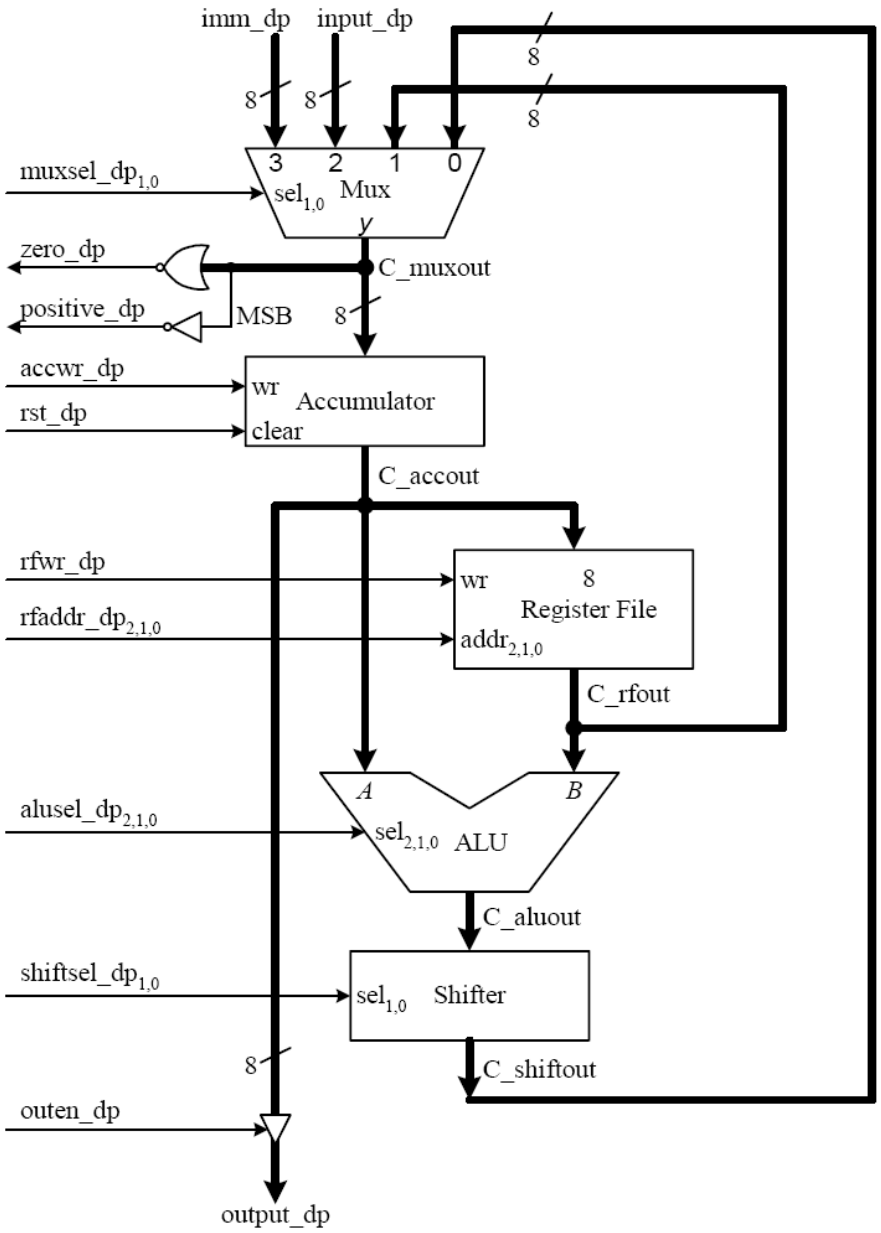

Fig 3.1. Datapath

3.3.2. Conditional Flags

The two conditional flags, zero and positive, are set by two comparators that check the value at the output of the mux which is the value that is to be written into the accumulator for these two conditions. To check for a value being zero, recall that just a NOR gate will do. In our case, we need an eight-input NOR gate because of the 8-bit wide data bus. To check for a positive number, we simply need to look at the most significant sign bit. A 2's complement positive number will have a zero sign bit, so a single inverter connected to the most significant bit of the data bus is all that is needed to generate this positive flag signal.

3.3.3. Accumulator

The accumulator is a standard 8-bit wide register with a write *wr* and clear *clear* control input signals. The write signal, connected to *accwr_dp*, is asserted whenever we want to write a value into the accumulator. The clear signal is connected to the main computer reset signal *rst_dp*, so that the accumulator is always cleared on reset. The content of the accumulator is always available at the accumulator output. The value from the accumulator is sent to three different places:

(1) It is sent to the output buffer for the OUT instruction;

(2) It is used as the first (A) operand for the ALU; and

(3) It is sent to the input of the register file for the STA instruction.

3.3.4. Register File

The register file has eight locations, each 8-bit wide. Three address lines, *rfaddr_dp2, rfaddr_dp1, rfaddr_dp0* are used to address the eight locations for both reading and writing. There are one read port and one write port. The read port is always active which means that it always has the value from the currently selected address location. However, to write to the selected location, the write control line *rfwr_dp* must be asserted before a value is written to the currently selected address location. Note that a separate read and write address lines is not required because all the instructions either perform just a read from the register file or a write to the register file. There is no one instruction that performs both a read and a write to the register file. Hence, only one set of address lines is needed for determining both the read and write locations.

3.3.5. ALU

The ALU has eight operations implemented as defined by the following table. The operations are selected by the three select lines *alusel_dp2*, *alusel_dp1*, and *alusel_dp0*. The select lines are asserted by the corresponding ALU instructions as shown under the *Instruction* column in the above table. The pass through operation is used by all non-ALU instructions.

alusel_dp2	alusel_dp1	alusel_dp0	Operation name	Operation	Instruction
0	0	0	pass	Pass A to output	Non-ALU
0	0	1	AND	A And B	AND A,rrr
0	1	0	OR	A OR B	OR A,rrr
0	1	1	NOT	A’	NOT A
1	0	0	Addition	A + B	ADD A,rrr
1	0	1	Subtraction	A - B	SUB A,rrr
1	1	0	Increment	A + 1	INC A
1	1	1	Decrement	A - 1	DEC A

Table 3.5. Alu operation

3.3.6. Shifter / Rotator

The Shifter has four operations implemented as defined by the following table. The operations are selected by the two select lines *shiftsel_dp*1, and *shiftsel_dp*0. The select lines are asserted by the corresponding Shifter/Rotator instructions as shown under the Instruction column in the above table. The pass through operation is used by all non-Shifter/Rotator instructions.

Shiftsel_dp1	Shiftsel_dp1	Operation	Instruction
0	0	Pass through	non Shift / Rotate instructions
0	1	Shift left and fill with 0	SHFL A
1	0	Shift right and fill with 0	SHFL A
1	1	Rotate right	ROTR A

Table 3.6. Shifter / Rotate operation

3.3.7. Output Buffer

The output buffer is a register with an enable control signal connected to *outen_dp*. Whenever the enable line is asserted, the output from the accumulator is stored into the buffer. The value stored in the output buffer is used as the output for the computer and is

always available. The enable line is asserted either by the OUT A instruction or by the system reset signal.

3.3.8. Control Word

From Figure 3.1, we see that the control word for this custom datapath has fourteen bits, which maps to the control signals for the different datapath components. These fourteen control signals are summarized in Table3.7.

Number	Signal name	Component	Purpose
14	muxsel_dp1	4-input mux	Select line1
13	muxsel_dp0	4-input mux	Select line1
12	accwr_dp	accumulator	Write enable
11	rst_dp	accumulator	clear
10	Rfwr_dp	register file	Write enable
9	Rfaddr_dp2	register file	Address line 2
8	Rfaddr_dp1	register file	Address line 1
7	Rfaddr_dp0	register file	Address line 0
6	alusel_dp2	ALU	Select line 2
5	alusel_dp1	ALU	Select line 1
4	alusel_dp0	ALU	Select line 0
3	shitsel_dp1	shifter	Select line 1
2	shitsel_dp0	shifter	Select line 0
1	outen_dp	Tri-state buffer	Output enable

Table 3.7. Control word signals for the Datapath

For example, to execute the ADD instruction, which adds the content of the accumulator with the content of the specified register file location and writes the result back into the accumulator, the value in the accumulator is passed to the A operand of the ALU. The B operand of the ALU comes from the register file, the location of which is selected from setting the register file address lines $rfaddr\_dp_{2,1,0}$. The appropriate ALU select lines $alusel\_dp_{2,1,0}$ are set to select the ADD operation. The shifter is not needed and so the pass

through operation is selected. The output of the shifter is routed back through input 0 of the multiplexer and finally written back to the accumulator.

So the control word for the instruction ADD A, 011 is

muxsel1	muxsel0	accwr	rst	rfwr	rfaddr2	rfaddr1	rfaddr0	alusel2	alusel1	alusel0
0	0	1	0	0	0	1	1	1	0	0

Shiftsel2	Shiftsel1	outen
0	0	0

3.4. CONTROL UNIT

The finite state machine for the control unit basically cycles through four main states: reset, fetch, decode, and execute, as shown in Figure 3.2. There is one execute state for each instruction in the instruction set.

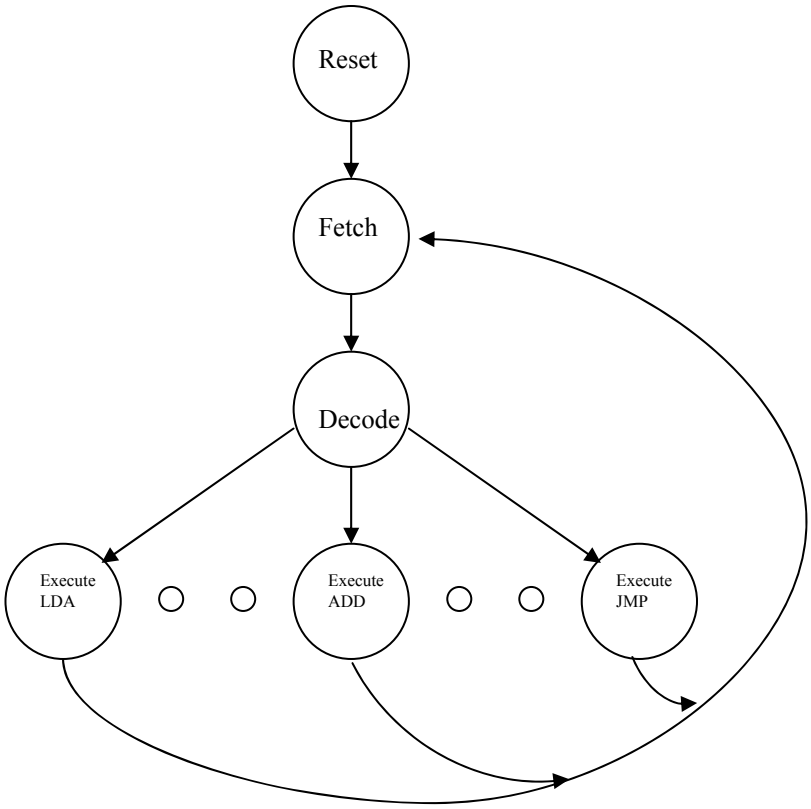

Fig.3.2. State diagram for the control unit.

3.4.1. Reset

The finite state machine starts executing from the reset state when the reset signal is asserted. On reset, the finite state machine initializes all its working variables and control signals. The variables include PC – program counter, IR – instruction register, state – the state variable. In addition, the content of the memory, i.e., the program for the computer to execute is also loaded at this time.

3.4.2. Fetch

In the fetch state, the memory content of the location pointed to by the PC is loaded into the instruction register. The PC is then incremented by one to prepare it for fetching the next instruction. If the fetched instruction is a jump instruction, then the PC will be changed accordingly during the execution phase.

3.4.3. Decode

The content that is stored in the instruction register is decoded according to the encoding that is assigned to the instructions as listed in table3.1, 3.2, 3.3, and 3.4. This is accomplished in VHDL using a CASE statement with the switch condition being the opcode. From the different cases, the state that is responsible for executing the corresponding instruction is assigned to the next state variable. As a result, the instruction will be executed starting at the beginning of the next clock cycle when the FSM enters this new state.

3.4.4. Execute

The execution state simply sets up the control word, which asserts the appropriate control signals for the datapath to carry out the necessary operations for executing a particular instruction. Each instruction, therefore, has its own execute state. For example, the execute state for the add instruction ADD A, 011 will set up the following control word.

muxsel1	muxsel0	accwr	rst	rfwr	rfaddr2	rfaddr1	rfaddr0	alusel2	alusel1	alusel0
0	0	1	0	0	0	1	1	1	0	0

Shiftsel2	Shiftsel1	outen
0	0	0

For all the jump instructions, no actions need to be taken by the datapath. It simply determines whether to perform the jump or not depending on the particular jump instruction and by checking on the zero and positive flags. If a jump is needed then the target address is calculated and then assigned to the PC. At the end of the execute state, the FSM goes back to the fetch state and the cycle repeats for the next instruction.

3.5. Complete Processor

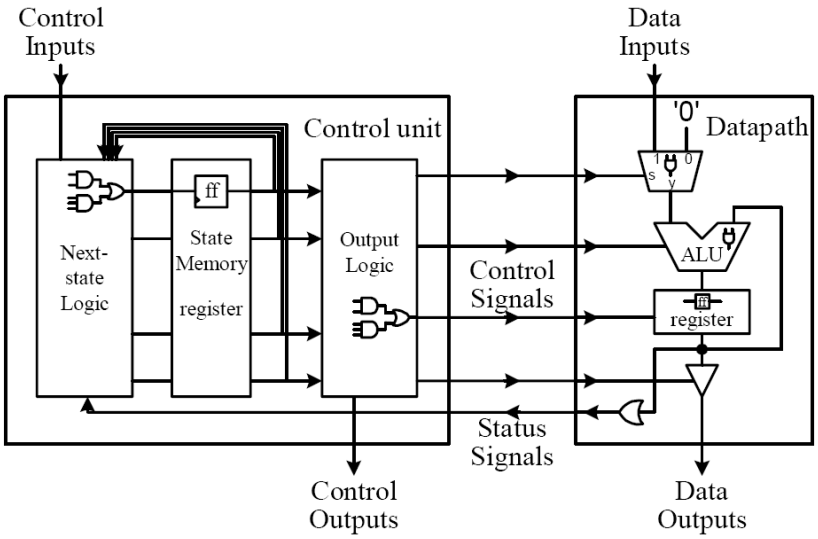

Fig.3.3. Complete general purpose processor

Chapter 4

DETERMINISTIC CLOCK GATING

4.1. INTRODUCTION

Deterministic clock-gating (DCG) is based on the key observation that for many of the pipeline stages in a modern processor, a circuit block usage in a specific cycle in the near future is *deterministically* known a few cycles ahead of time. DCG exploits this advance knowledge to clock-gate the unused blocks. In particular, we propose to clock-gate execution units, pipeline latches of back-end stages after issue, L1 *D*-cache word-line decoders, and result bus drivers. In an out-of-order pipeline, whether these blocks will be used is known at the *end of issue* based on the instructions issued. There is *at least one cycle* of register read stage between issue and the stages using execution units, *D*-cache word-line decoder, result bus driver, and the back-end pipeline latches. DCG exploits this one-cycle advance knowledge to clock-gate the unused blocks without impacting the clock speed.

DCG has the following key features.

1) DCG is based on actual usage of circuit blocks and not on predictions. Therefore, DCG avoids performance loss due to mispredictions causing circuits to be gated when they are needed, and lost opportunity due to mispredictions causing circuits not to be gated when they are idle.

2) DCG clock-gates at fine granularities of a few (1–2) cycles on small circuit blocks (execution units, *D*-cache decoders, result bus drivers, and pipeline latches). The fine granularity enables flexible gating of individual pipeline stages without the all-or-nothing restriction of gating the entire pipeline backend, making DCG effective. However, DCG's blocks are still substantially larger than the few gates added for clock-gating, allowing DCG to amortize the overhead.

3) DCG is a simple technique requiring no fine-tuning of thresholds, and is general enough to be applicable to clustered and non-clustered micro architectures.

DCG not only achieves large power savings, but also incurs no performance loss, while being simple.

In this section, we analyze the opportunity of DCG in different parts of a superscalar micro architecture.

DCG depends on two factors:

1) Opportunity due to existence of idle clock cycles (i.e., cycles when a logic block is not being used) and

2) Advance information about when the logic block will not be used in the future.

Fig. 4.1 depicts the general pipeline model for a superscalar processor. The pipeline consists of eight stages with pipeline latches between successive stages, used for propagating instruction/ data from one stage to the next. Here is the explanation, why we do or do not clock-gate each individual pipeline latch and stage.

4.2 DCG FOR PIPELINE LATCHES

Pipeline latches unconditionally latch their inputs at every clock edge, resulting in high power dissipation. As the technology scales down, deeper pipeline stages with more latches are used. Furthermore, the data width (e.g., 32 versus 64 b) also increases with microprocessor evolution. Consequently, the ratio of the latch power to the total processor power increases. Because most of the stage latches have some idle cycles, clock-gating the latches during these cycles can substantially save processor power. Each of the stages to determine if an idle cycle for the stage can be known in advance is analyzed.

Instructions are fetched from the instruction cache every cycle. The instructions are then decoded, checked for dependences, renamed, and deposited in an instruction window. For a branch, the instructions on the predicted path will be taken before the branch is resolved. At the end of decode, we can determine how many of the instructions are in the predicted path out of those fetched. That is, if the third instruction in a fetched block is a branch and the branch is predicted to be taken then the instructions from the fourth instruction to the end of the fetched block are thrown away. Only the first three instructions enter the rename stage.

Unfortunately, we cannot clock-gate the latches following fetch and decode because before decode we do not know how many instructions are in the fetched path. However, we can determine the number of instructions that will enter the rename stage at the end of decode and clock-gate the unnecessary parts of the rename latch. We have the entire rename stage to set up the clock-gate control of the rename latch.

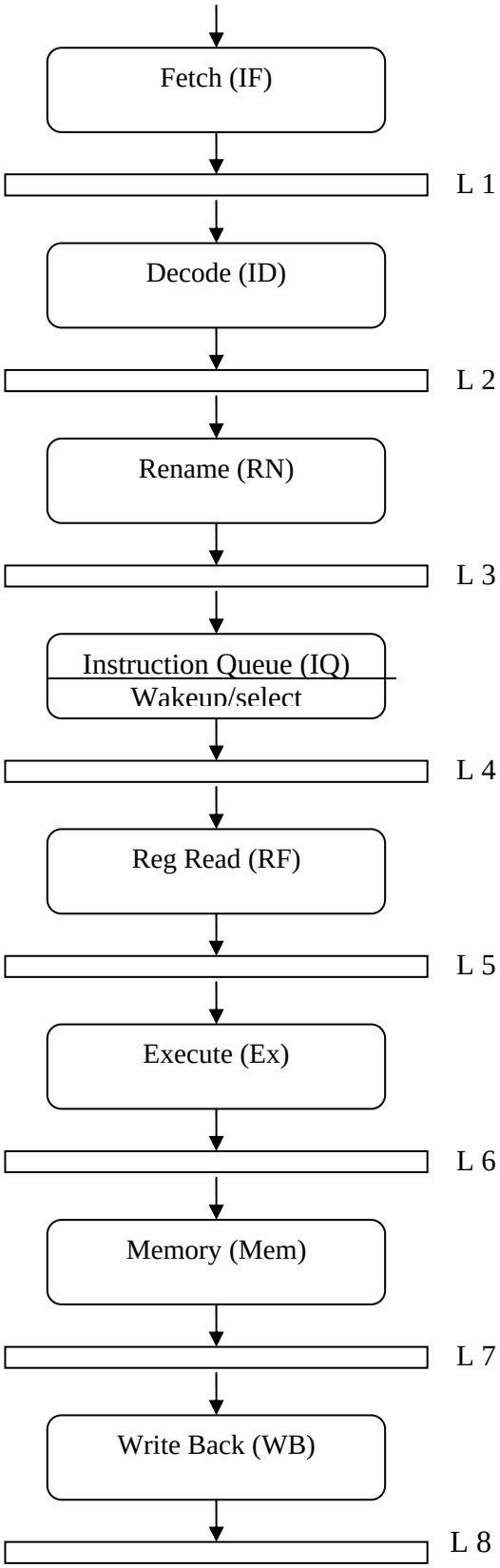

Fig 4.1 basic superscalar pipeline

Because we can identify which and how many instructions are selected to issue only at the very end of issue, we do not have enough time to clock-gate the issue latch. We can clock-gate the latches for the rest of the pipeline stages [register read (RF), execute (Ex), memory access (Mem) and Write-Back (WB)]. At the beginning of the each of the stages we know how many instructions are entering the stage, and we can exploit the time during the stage to set up the clock-gate control for these latches.

4.3 DCG FOR PIPELINE STAGES

Fetch stage uses the decoders in the instruction cache and decode stage uses instruction decoder, both of which are often implemented with dynamic logic circuits. However, we cannot clock-gate fetch and decode logic, because fetch and decode occur almost every cycle. We do not know which instructions are useless until we decode them, which is too late to clock-gate the decode stage. Rename stage consumes little power and so we do not consider rename stage for clock-gating.

The issue stage consists of the issue queue, which uses an associative array and a wakeup/select combinational logic. Issue queue entries that are either deterministically determined to be empty, or deterministically known to be already woken up, are essentially clock-gated.

Register read stage consists of a register file implemented using an array. However, only at the very end of issue, we know how many instructions are selected and are going to access the register file in the next cycle. Hence, there is no time to clock-gate the register file.
We can clock-gate the execution units, which are often implemented with dynamic logic blocks for high performance. Based on the instructions issued, we deterministically know at the end of issue which unit is going to be used in the cycle after the register read stage. Hence, we can clock-gate the rest of the unused execution units, by setting the clock-gate control during the read cycle.

Modern caches use dynamic logic for word-line decoding and the write-back stage uses result bus driver to route result data to the register file. Instructions that enter the execute stage go through the memory and write-back stages. We can use the same clock-gate control used in execute to clock-gate the relevant logic in these stages. The control signal needs to be delayed by one and two clock cycle(s), respectively, for the memory and write-back stages.

4.4. IMPLEMENTATION OF DCG

4.4.1. Execution Units

At the end of instruction issue, we know which execution units will be used in the execute stage, a few cycles into the future. The selection logic in a conventional issue queue not only selects which instructions are to be issued based on execution unit availability, but also matches instructions to execution unit. Hence, we leverage the selection logic to provide information about which execution units will remain unused and clock-gate those units.

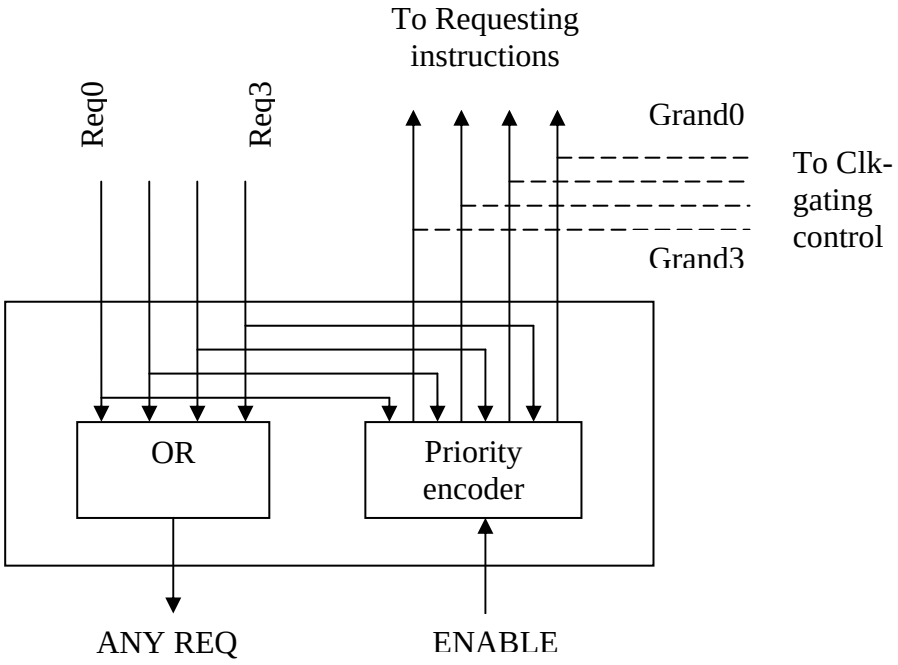

Fig 4.2. Schematic of a selection logic cell with the clock-gate signals extracted from it.

Fig. 4.2 shows the schematic of selection logic associated with one type of execution units (e.g., integer ALU, or floating-point adder, or floating-point multiplier, etc.). The request signals (REQ) come from the ready instructions once the wakeup logic determines which instructions are ready. The selection logic uses some selection policy to select a subset of the ready instructions, and generates the corresponding grant signals (GRANT). In our implementation, we send the GRANT signals to the clock-gate control.

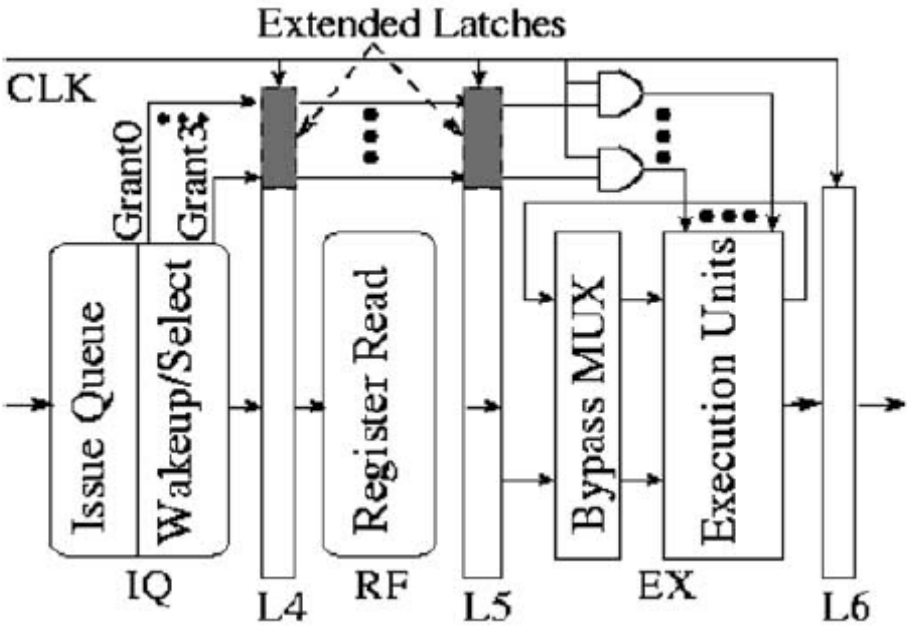

Fig 4.3. Clock-gating of the execution units.

Fig. 4.3 shows the pipeline details of the control. Because instructions selected in cycle X use the execution units in cycle (as shown in Fig. 4.4), we have to pass the GRANT signals down the pipeline through latches for proper timing of clock-gating. We extend the pipeline latches for the issue and read stages by a few extra bits to hold the GRANT signals. We note that the gated clock line (output of the AND gates in Fig. 4.3) that feeds the execution units may be skewed a bit because of the delay through the latch and the AND gate. This skew affects only the *precharge* phase and not the *evaluate* phase. Therefore, DCG is likely not to lengthen execution unit latencies.

The control for clock-gating execution units is simple and the overhead of the extended latches and the AND gates is small compared to the execution units (e.g., 32- or 64-b carry-look ahead adders) themselves. Therefore, the area and power overhead of the control circuitry are easily amortized by the significant power savings achieved. If execution units keep toggling between gated and non-gated modes, the control circuitry keeps switching, resulting in an increased overhead due to the power consumed by the control circuitry. To alleviate this problem, we apply *sequential priority policy* for execution units: Among the execution units of the same type, we statically assign priorities to the units, so that the higher priority units are always chosen to be used before the lower priority units. Thus, most of the

time the (lower) higher priority units stay in (gated) non-gated mode, minimizing the control power overhead.

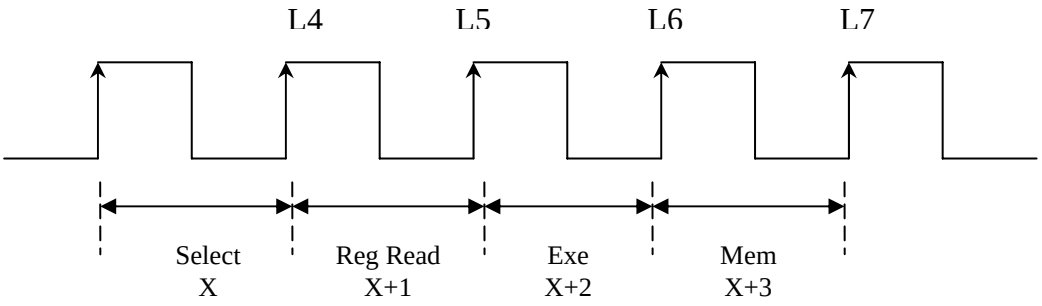

Fig 4.4. Timing diagram for execution units clock-gating.

4.4.2. Pipeline Latches

We clock-gate pipeline latches at the end of rename, register read, execute, memory, and write-Back stages. For rename, the number of clock-gated latches in any cycle is known from the decode stage in the previous cycle. For latches in the other stages, the number of clock-gated latches in any cycle is known from the issue stage. We augment the issue stage to generate a one-hot encoding of how many instructions are issued every cycle. The encoding has a “0” to represent an empty issue slot, and a “1” to represent a full issue slot for an issued instruction, for all the issue slots of the pipeline. Much like the execution units, the clock the one-hot encoding is passed down the pipe via extended pipeline latches.

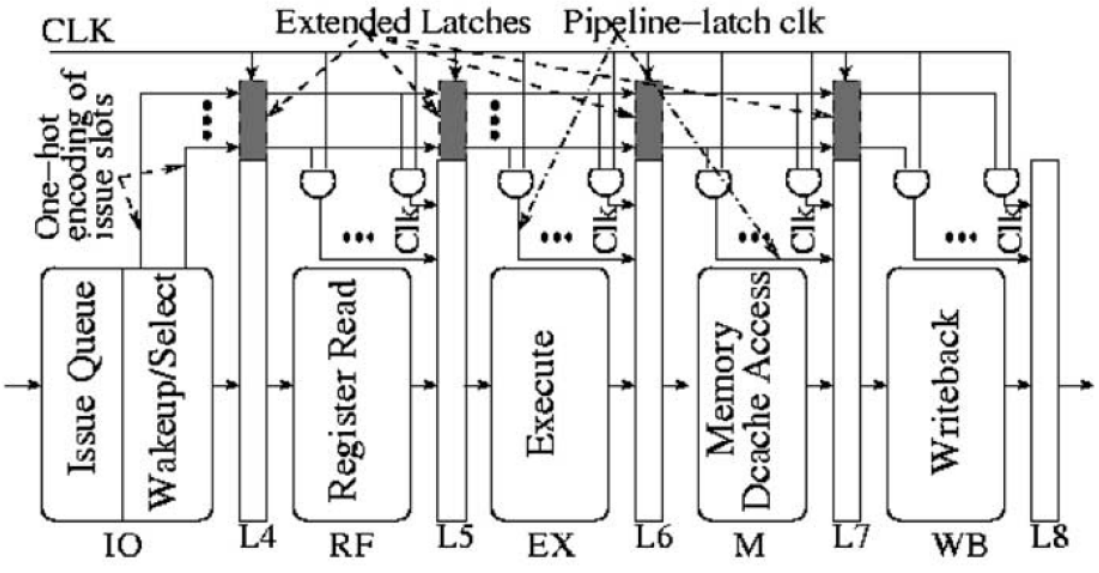

Fig 4.5. Clock-gating of pipeline latches.

Fig. 4.5 shows the clock-gating control for the stages following issue queue. The outputs of the extended latches carrying the one-hot encoding are ANDed with the clock line to generate a set of gated clock inputs for pipeline latches corresponding to individual issue slots. Note that the clock line for the extra latches themselves is not gated.

Extensions to the pipeline latches and the extra AND gates for the control are small compared to the pipeline latches (containing issue-width X number of operands per instruction X operand width bits, e.g., 8 X 2 X 64 = 1024 b) themselves, and clock drivers, respectively. Hence, the impact of the extra control logic on area and power is not significant.

Chapter 5

APPLICATION

5.1. TRAFFIC LIGHT CONTROLLER

The controller to be designed controls the Traffic lights of a junction intersecting two main roads. Figure 5.1 shows the location of the Traffic lights. Sensors at the intersection detect the presence of cars on the highway and side road.

The controller operates the traffic lights at an intersection where two-way street running north and south intersects a two-way street running east and west. The goal is to design the controller so that collisions are avoided, and no traffic waits at a red light forever.

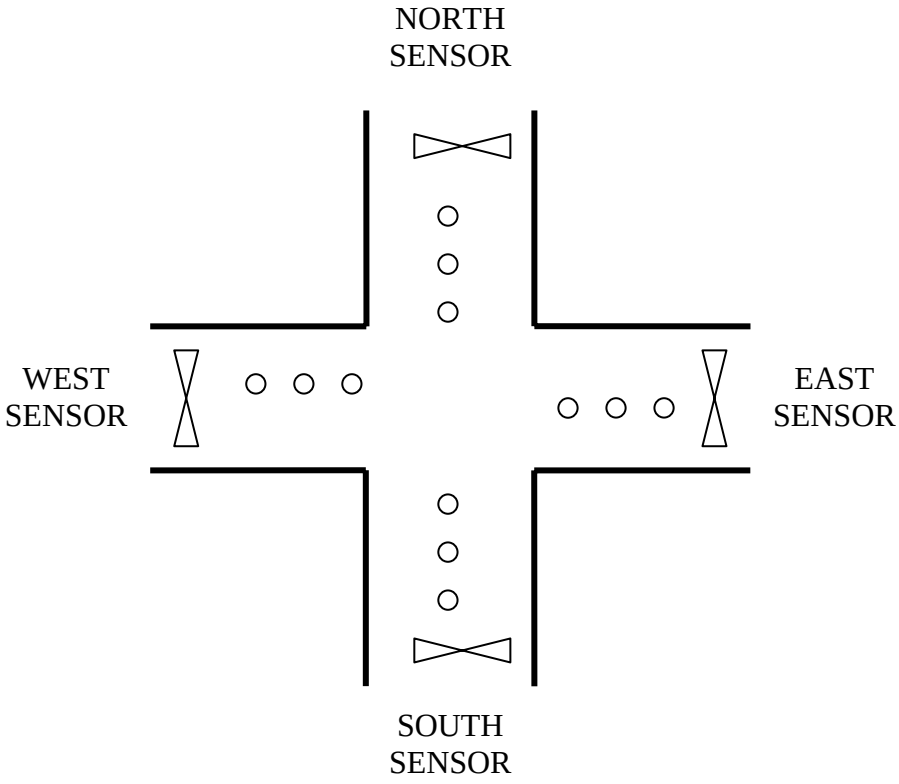

Fig 5.1 Traffic light controller

The controller has four traffic sensor inputs, N_Sensor, W_Sensor, S_Sensor and E_Sensor indicating when a vehicle is present at the intersection traveling in the north, south, west and east directions respectively. There are four green outputs, `N_G`, `S_G`, `W_G` and `E_G`, indicating that a green light should be given to traffic in each of the four directions.

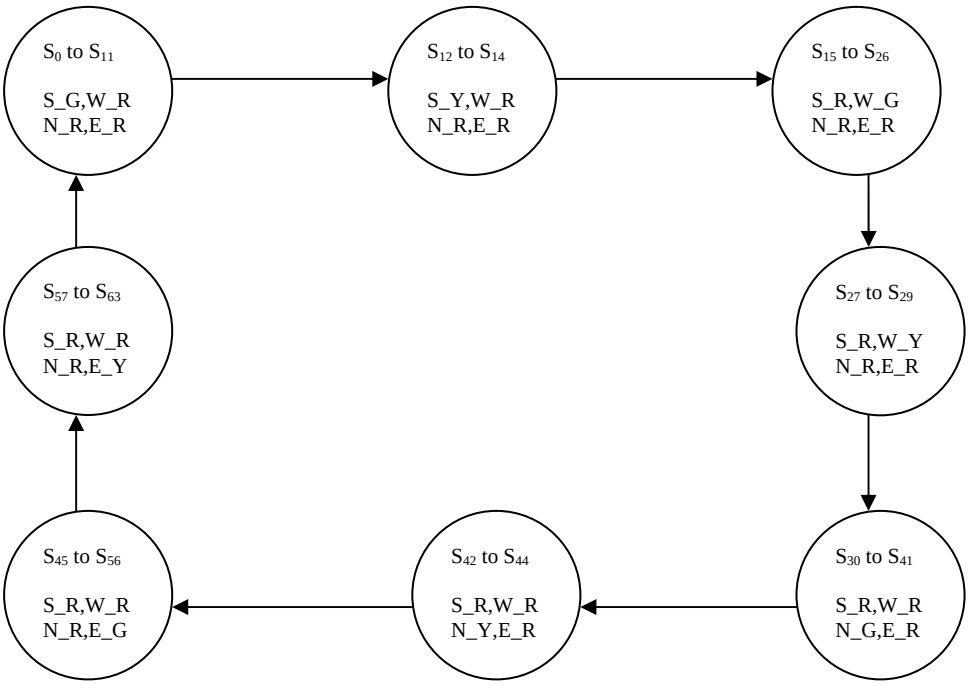

Fig 5.2 state diagram of traffic light controller

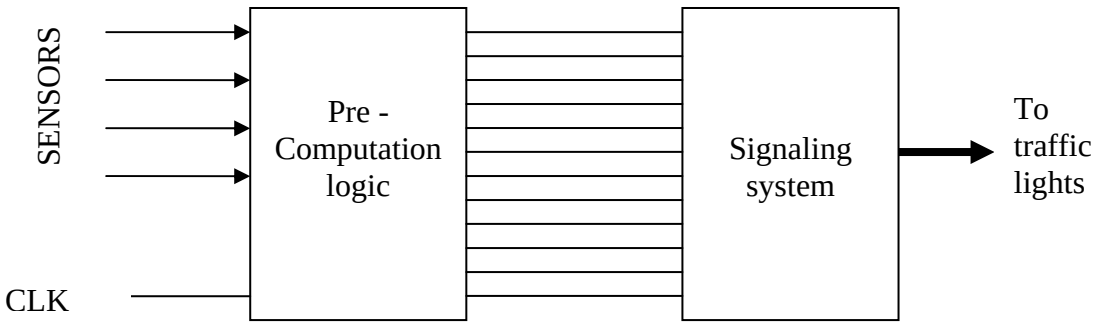

Fig 5.3 Pre-computation based traffic light controller

When any sensor is ON, switch on the respective green light and change the state of the present green light. Note that, these two assignments should occur simultaneously. Otherwise, there may be a collision of traffic a junction.

The pre computation logic should be designed in such a way that the resulted logic is to be work as a normal traffic light controller and switch to corresponding route depending on the sensors..

Effective clock-gating, however, requires a methodology that determines which signals are to be gated, when, and for how long. Clock-gating schemes that either result in frequent toggling of the signals between enabled and disabled states, or apply clock-gating to such small blocks that the clock-gating control circuitry is almost as large as the blocks themselves, incur large overhead. This overhead may result in power dissipation to be higher than that without clock-gating.

Chapter 6

SIMULATION RESULTS

6.1. HARDWARE DESCRIPTION LANGUAGE

The two most popular hardware description languages are VHDL and Verilog. The HDL used for our thesis is VHDL.

VHDL is a hardware description language used to describe the behavior and structure of digital systems. The acronym VHDL stands for VHSIC Hard ware Description Language, and VHSIC in turn stands for very high speed integrated circuit. However, VHDL is a general purpose hardware description language, which can be used to describe and simulate the operation of a wide variety of digital systems, ranging in complexity from a few gates to an interconnection of many complex integrated circuits. The VHDL language is widely used in industry.

VHDL can describe a digital system at several different levels- behavioral, dataflow and structural. VHDL leads naturally to a top-down design methodology, in which the system is first specified at a high level and tested using a simulator. After the system is debugged at this level the design can gradually be refined eventually leading to structural description.
The language has the following feature:

- Designs may be decomposed hierarchically
- Each designs element has both a well-defined interface (for connecting in it other elements) and a precise behavioral specification (for simulating it).
- Behavioral specifications can use either an algorithm or an actual hardware structure to define an element’s operation.
- Concurrency timing and clocking can all be modular. VHDL handles asynchronous as well as synchronous sequential- circuit structures.
- The logical operation and timing behavior of a design can be simulated.

While the VHDL language and simulation environment were important innovations by themselves, VHDL’s utility and popularity took a quantum leap with the commercial development of VHDL synthesis tools. These programs can create logic – circuit structures directly from VHDL behavioral description using VHDL, simulate and synthesize anything from a simple combinational circuit to a complete microprocessor system on chip.

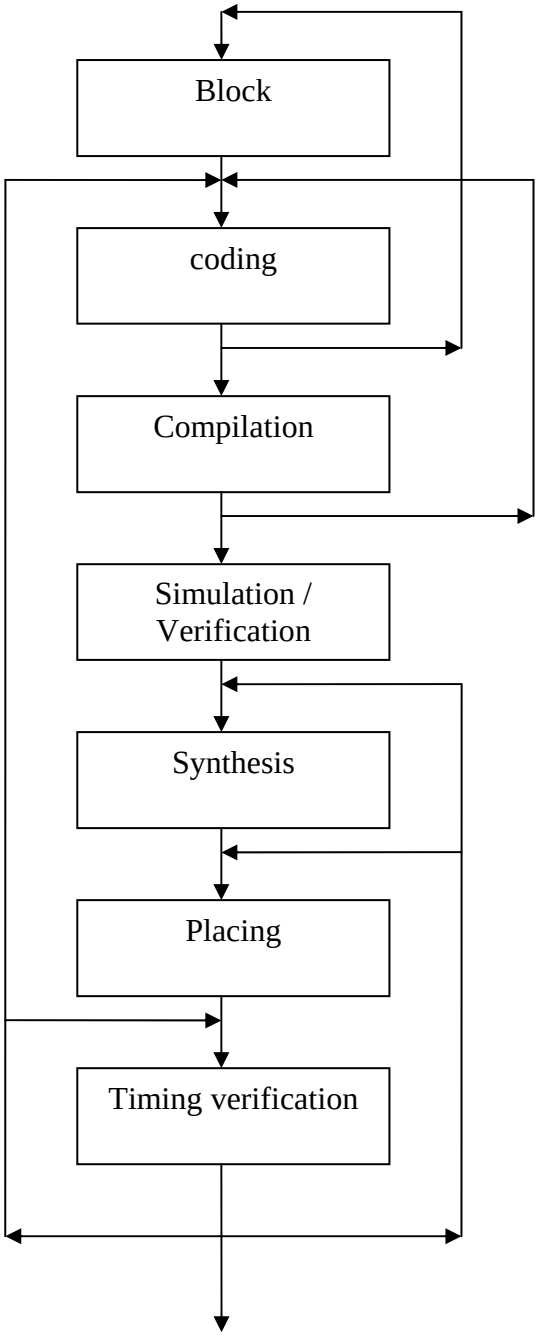

Fig.6.1. HDL Based design flow

A programmable logic device or PLD is an electronic component used to build digital circuits. Unlike a logic gate, which has a fixed function, a PLD has an undefined function at the time of manufacture. Before the PLD can be used in a circuit it must be programmed. Programmability of logic means that new chip designs can be tested and easily changed without incurring the huge photo mask costs for chips completed in a semiconductor fab. In addition, memory-based PLDs can be reprogrammed over and over. Figure 6.1 contains a block

diagram of a hypothetical CPLD. Each of the four logic blocks shown there is the equivalent of one PLD. However, in an actual CPLD there may be more (or less) than four logic blocks. Note also that these logic blocks are themselves comprised of macro cells and interconnect wiring, just like an ordinary PLD.

Unlike the programmable interconnect within a PLD, the switch matrix within a CPLD may or may not be fully connected. In other words, some of the theoretically possible connections between logic block outputs and inputs may not actually be supported within a given CPLD. The effect of this is most often to make 100% utilization of the macro cells very difficult to achieve. Some hardware designs simply won't fit within a given CPLD, even though there are sufficient logic gates and flip-flops available.

Because CPLDs can hold larger designs than PLDs, their potential uses are more varied. They are still sometimes used for simple applications like address decoding, but more often contain high-performance control-logic or complex finite state machines. At the high-end (in terms of numbers of gates), there is also a lot of overlap in potential applications with FPGAs. Traditionally, CPLDs have been chosen over FPGAs whenever high-performance logic is required. Because of its less flexible internal architecture, the delay through a CPLD (measured in nanoseconds) is more predictable and usually shorter.

6.2. FIELD PROGRAMMABLE GATE ARRAYS

Field Programmable Gate Arrays (FPGAs) can be used to implement just about any hardware design. There are three key parts of its structure: logic blocks, interconnect, and I/O blocks.
The I/O blocks form a ring around the outer edge of the part. Each of these provides individually selectable input, output, or bi-directional access to one of the general-purpose I/O pins on the exterior of the FPGA package. Inside the ring of I/O blocks lies a rectangular array of logic blocks. And connecting logic blocks to logic blocks and I/O blocks to logic blocks is the programmable interconnect wiring.
The FPGA used for testing is VIRTEX-II.

VIRTEXII PRO FPGA

Configurable Logic Blocks (CLBs):

CLB resources include four slices and two 3-state buffers.

Each slice is equivalent and contains:

- Two function generators (F & G)
- Two storage elements
- Arithmetic logic gates
- Large multiplexers
- Wide function capability
- Fast carry look-ahead chain
- Horizontal cascade chain (OR gate)

Configuration

Virtex-II Pro devices are configured by loading the bit stream into internal configuration memory using one of the following modes:

- Slave-serial mode
- Master-serial mode
- Slave Select MAP mode
- Master Select MAP mode
- Boundary-Scan mode (IEEE 1532)

Ordering information of VIRTEX-II is **XC2VPX20 -5 FF 896 C**

- **XC2VPX20** - Device Type
- **-6** -- Speed Grade (-5 , -6)
- **FF** -- Package Type
- **896** -- No. of pins
- **C** -- Temperature Range

6.3. SIMULATION RESULTS OF GENERAL PURPOSE PROCESSOR

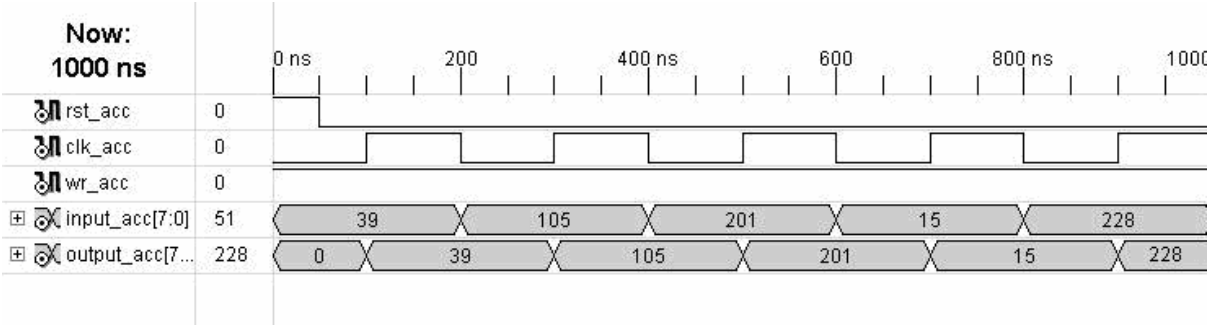

Fig.6.2. Accumulator output

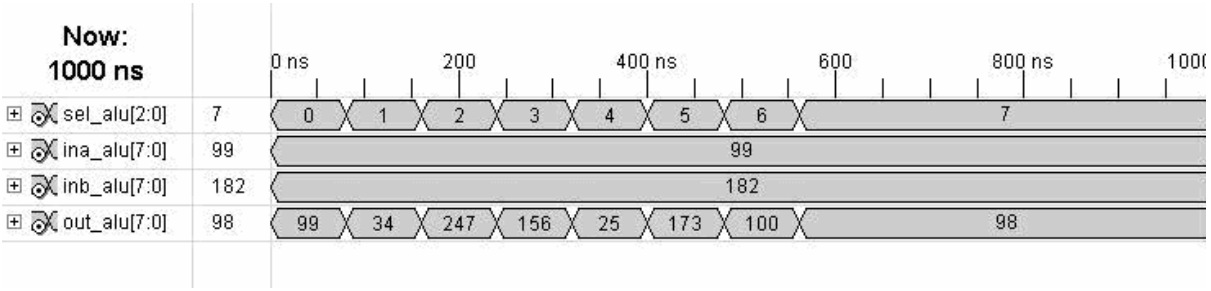

Fig.6.3. ALU output when the inputs are A = “01100011” and B = “10110110”.

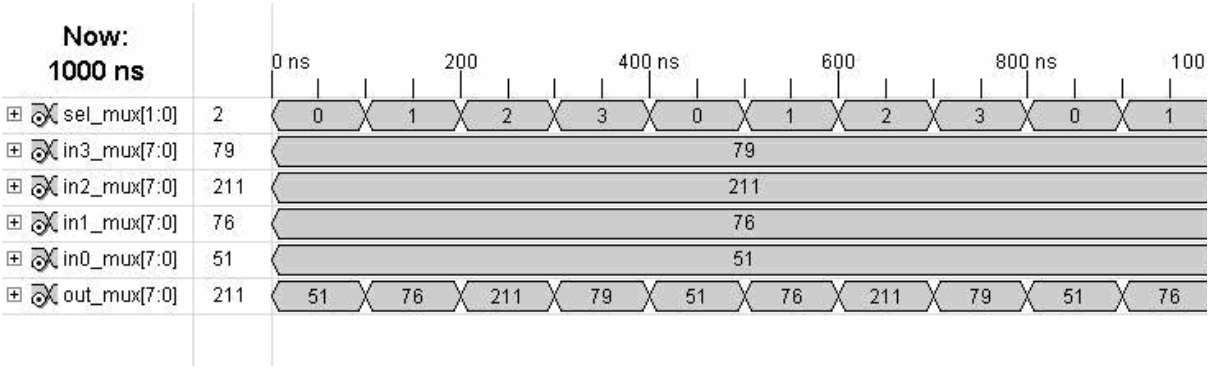

Fig.6.4. Multiplexer output.

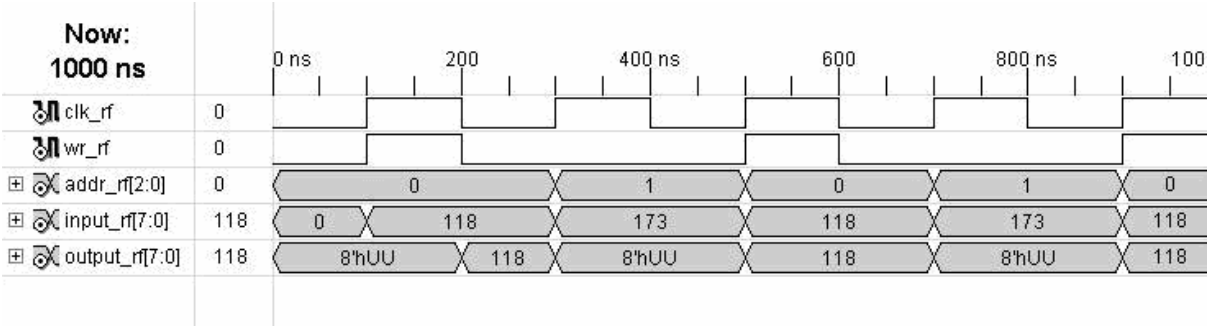

Fig.6.5. Regfile output.

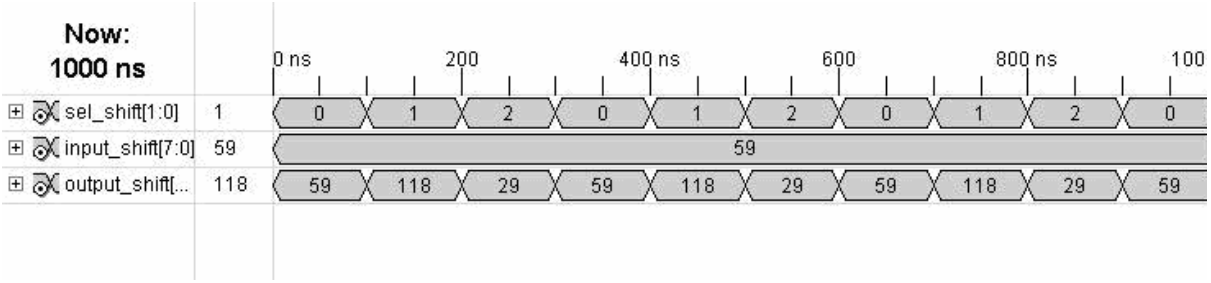

Fig.6.6. Shifter output when the input is “00111011”.

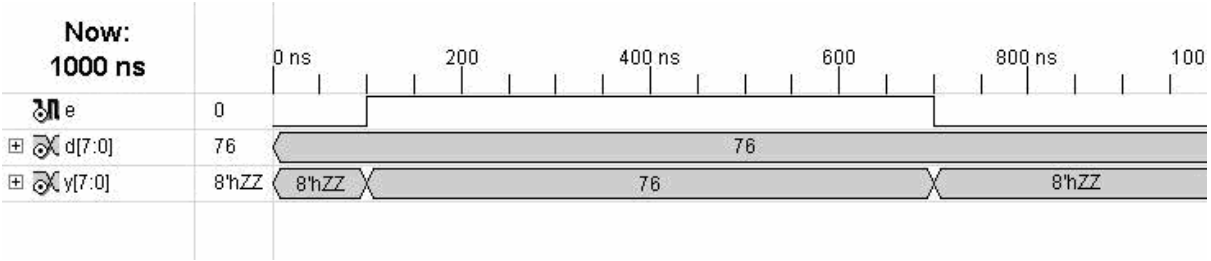

Fig.6.7. Tri-state buffer output.

6.4. SIMULATION RESULTS OF TRAFFIC LIGHT CONTROLLER

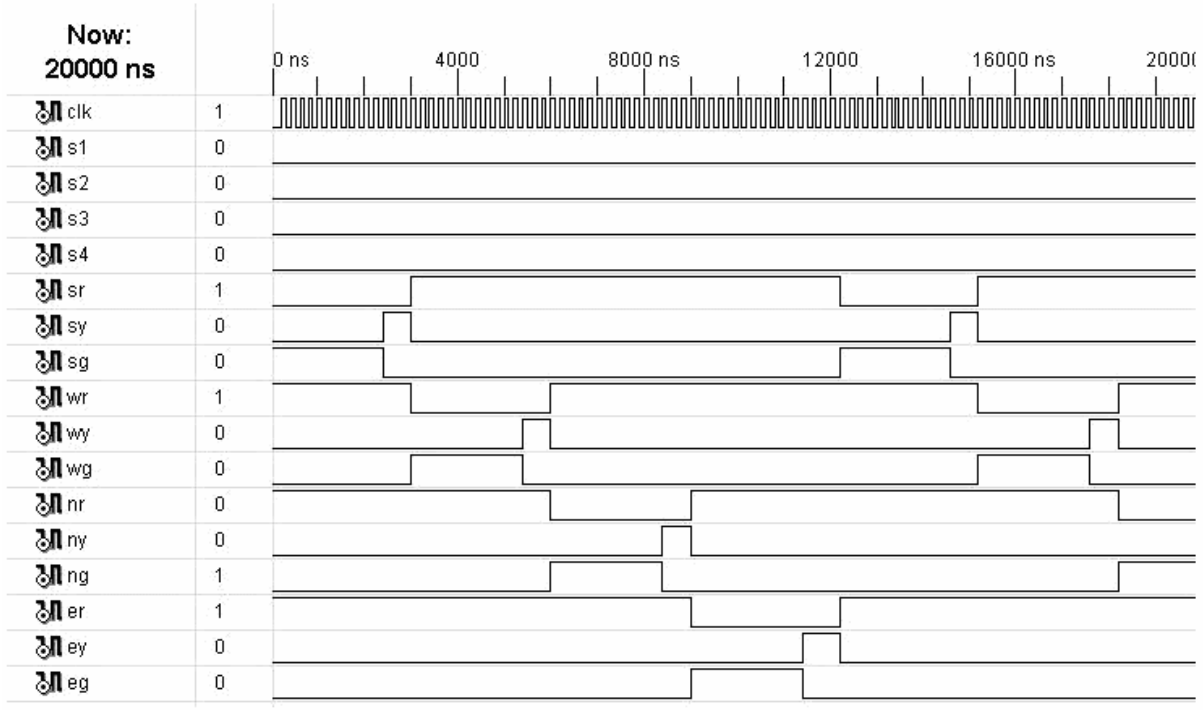

Fig.6.8.Normal Traffic light controller.

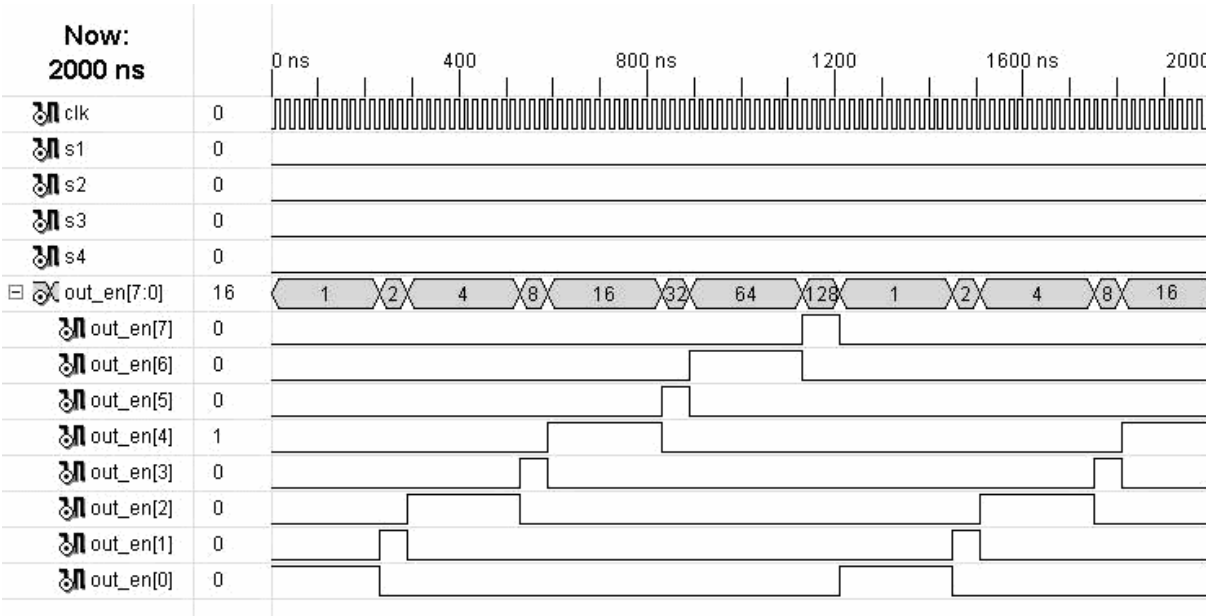

Fig.6.9. Pre-computation logic DCG Traffic light controller output.

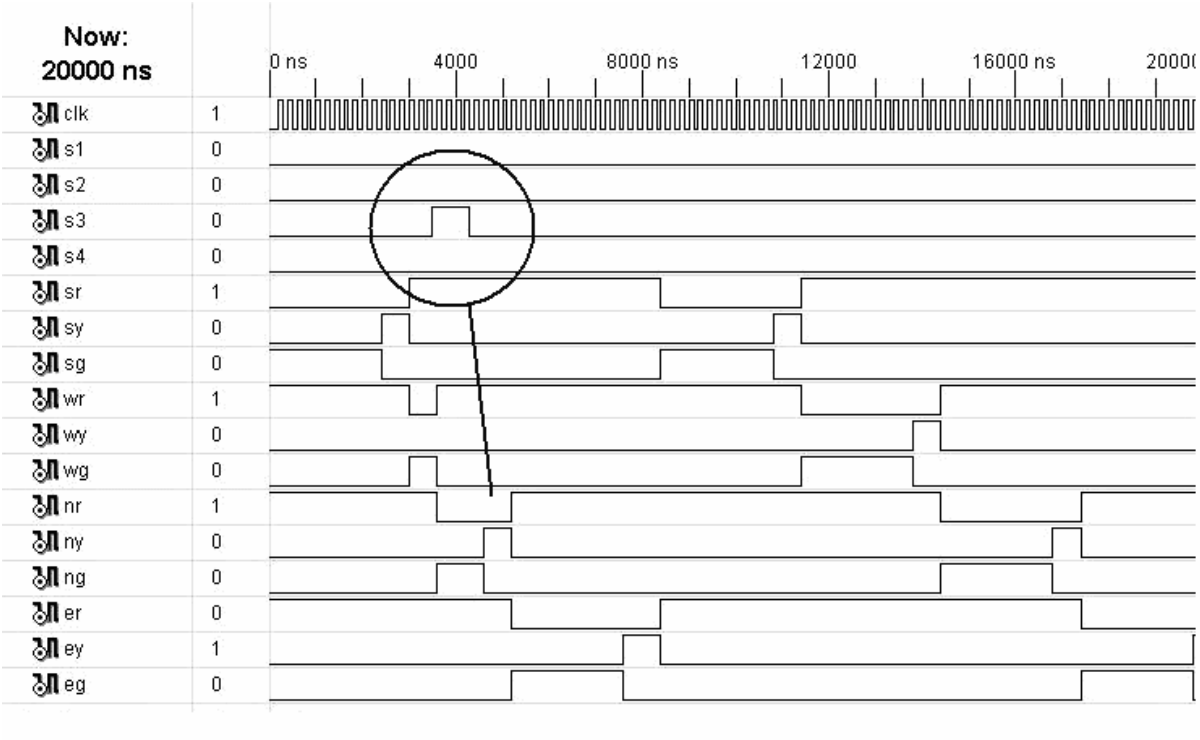

Fig.6.10. PCL based DCG Traffic light controller output when N_sensor ON.

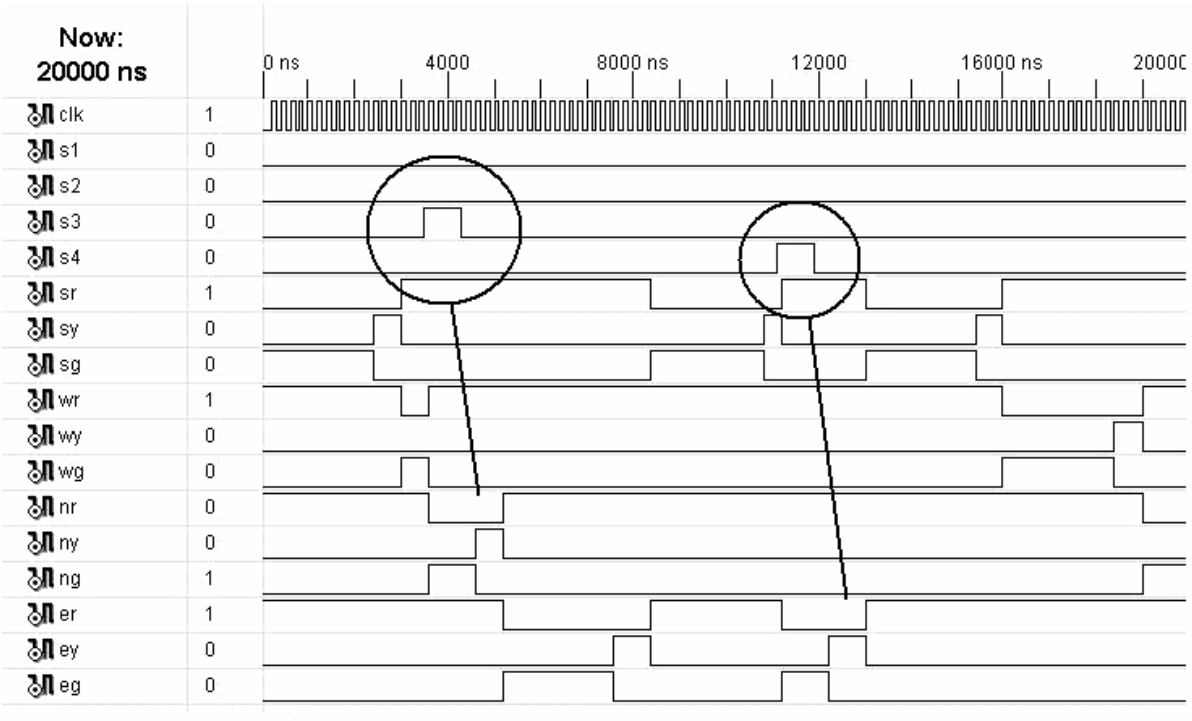

Fig.6.11. PCL based DCG Traffic light controller output TWO sensors ON.

6.5. POWER CALCULATIONS

	ALU	ACC	MUX	Shifter	Tri-state buffer
No.of slices	18	9	8	8	1
No. of 4 i/p LUT's	25	9	16	9	1
No. of IO's	27	19	42	18	17
No. of IO Buffers	27	19	42	18	17

Table 6.1 Cell Usage for the General Purpose Processor

	ALU	ACC	MUX	Shifter	Tri-state buffer
No.of slices	13	6	5	5	1
No. of 4 i/p LUT's	19	6	13	6	1
No. of IO's	23	16	35	14	13
No. of IO Buffers	23	16	35	14	13

Table 6.2 Cell Usage for the General Purpose Processor after DCG applied.

	Traffic light controller	Traffic light controller after PCL DCG
No.of slices	20	18
No. of 4 i/p LUT's	40	37
No. of IO FF's	12	8
No. of IO Buffers	18	14
No.of slice FF's	28	23
Timing Delay	4.470 ns	3.565 ns
Power consumption	330 uW	219 uW

Table 6.3 Cell Usage for the Traffic light controller.

Chapter 7

CONCLUSION

CONCLUSION

Deterministic clock-gating (DCG) methodology is based on the key observation that for many of the stages in a modern pipeline, a circuit block's usage in a specific cycle in the near future is deterministically known a few cycles ahead of time. Using this advance information, DCG clock-gates unused execution units, pipeline latches.

Results show that DCG is very effective in reducing clock power. 25 – 33 % power consumption is reduced by using this method. As high-performance processor pipelines get deeper and power becomes a more critical factor, DCG's effectiveness and simplicity will continue to be important.

Effective clock-gating, however, requires a methodology that determines which circuits are gated, when, and for how long. Care to be taken while designing the clock-gating control circuitry; otherwise the circuitry may become an overhead. This overhead may result in power dissipation to be higher than that without clock-gating.

REFERENCES

1. Bipul C. Paul, Amit Agarwal, Kaushik Roy, “Low-power design techniques for scaled technologies”, INTEGRATION, the VLSI journal, science direct 39(2006).

2. L. Hai, S. Bhunia, Y. Chen, K. Roy, T.N. Vijay Kumar, “ DCG : deterministic clock gating for low-power microprocessor design ”, IEEE Trans. VLSI Syst. 12 (2004), pp.245-254.

3. Yan Luo , Jia Yu , Jun Yang , Laxmi Bhuyan, Low power network processor design using clock gating, Proceedings of the 42nd annual conference on Design automation, June 13-17, 2005, San Diego, California, USA.

4. Hai Li, Swarup Bhunia, Yiran Chen, T. N. Vijaykumar, and Kaushik Roy, “Deterministic Clock Gating for Microprocessor Power Reduction”,1285 EE Building, ECE Department, Purdue University.

5. Enoch O. Hwang, “Microprocessor Design Principles and Practices”, Brooks / Cole, 2004.

6. Hai Li , Chen-Yong Cher , Kaushik Roy , T. N. Vijaykumar, Combined circuit and architectural level variable supply-voltage scaling for low power, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, v.13 n.5, p.564-576, May 2005.

7. D. Folegnani and A. Gonzalez, “Energy-effective issue logic”, in Proc.28th Int. Symp. Computer Architecture (ISCA), July 2001, pp. 230 - 239.

8. D. Garrett, M. Stan, A. Dean, “Challenges in clock gating for a low power ASIC methodology”, in International Symposium on Low Power Electronics and Design, 1999, pp. 176 - 181.

9. J. Oh, M. Pedram, “Gated clock routing for low-power microprocessor design”, IEEE Trans. Comput. Aided Des.Integr. Circuits Syst. 20 (2001) 715 – 722.

10. N. Raghavan, V. Akella, S. Bakshi, “Automatic insertion of gated clocks at register transfer level”, in: International Conference on VLSI Design, 1999, pp. 48 – 54.

11. L. Benini, G.D. Micheli, “Automatic synthesis of low power gated clock finite state machines”, IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 15 (1996) 630 – 643.

12. R. I. Bahar and S. Manne, “Power and energy reduction via pipeline balancing,” in Proc. 28th Int. Symp. Computer Architecture (ISCA), July 2001, pp. 218–229.

13. D. Brooks, V. Tiwari, and M. Martonosi, “Wattch: a framework for architectural-level power analysis and optimizations,” in Proc. 27th Int.Symp. Computer Architecture (ISCA), July 2000, pp. 83–94.

14. S. Palacharla, N. P. Jouppi, and J. E. Smith, “Complexity-effective superscalar processors,” in Proc. 24th Annu. Int. Symp. Computer Architecture (ISCA), June 1997, pp. 206–218.

15. S. Manne, A. Klauser, and D. Grunwald, “Pipeline gating: speculation control for energy reduction,” in Proc. 25th Int. Symp. Computer Architecture (ISCA), June 1998, pp. 132–141.

16. D. Folegnani and A. Gonzalez, “Energy-effective issue logic,” in Proc. 28th Int. Symp. Computer Architecture (ISCA), July 2001, pp. 230–239.

17. D. Brooks and M. Martonosi, “Value-based clock gating and operation packing: dynamic strategies for improving processor power and performance,” ACM Trans. Comput. Syst., vol. 18, no. 2, pp. 89–126, May 2000.

18. J. C. Monteiro, “Power optimization using dynamic power management,” in Proc. XII Symp. Integrated Circuits Systems Design (ICSD), Sept. 1999, pp. 134–139.

19. Massoud Pedram “Power Minimization in IC Design: Principles and Applications”, Department of EE-Systems, University of Southern California, Los Angeles CA.

20. William M. Johnson “Super-Scalar Processor Design”, Computer Systems Laboratory, Stanford University Stanford, CA, June 1989.

21. Ricardo E. Gonzalez “LOW-POWER PROCESSOR DESIGN”, Computer Systems Laboratory, Stanford University Stanford, CA, June 1997.

22. Qing WU “Clock-Gating and Its Application to Low Power Design of Sequential Circuits”, Department of Electrical Engineering-Systems, University of Southern California, Los Angeles, CA, USA.

APPENDIX

Device utilization summary for complete processor:

Selected Device : 2vp4ff672-5

Number of Slices:	140	out of	3008	4%
Number of Slice Flip Flops:	77	out of	6016	1%
Number of 4 input LUTs:	260	out of	6016	4%
Number used as logic:	252			
Number used as RAMs:	8			
Number of IOs:	18			
Number of bonded IOBs:	18	out of	348	5%
Number of GCLKs:	1	out of	16	6%

Timing Summary:

Speed Grade: -5

Minimum period: 10.131ns (Maximum Frequency: 98.705MHz)

Minimum input arrival time before clock: 4.902ns

Maximum output required time after clock: 5.090ns

Maximum combinational path delay: 6.131ns

Device utilization summary for Datapath:

Selected Device : 2vp4ff672-5

Number of Slices:	76	out of	3008	2%
Number of Slice Flip Flops:	16	out of	6016	0%
Number of 4 input LUTs:	137	out of	6016	2%
Number used as logic:	129			
Number used as RAMs:	8			

Number of IOs:	41			
Number of bonded IOBs:	41	out of	348	11%
Number of GCLKs:	1	out of	16	6%

Timing Summary:

Speed Grade: -5

Minimum period: 8.613ns (Maximum Frequency: 116.100MHz)

Minimum input arrival time before clock: 9.735ns

Maximum output required time after clock: 13.396ns

Maximum combinational path delay: 14.518ns

Device utilization summary for control unit:

Selected Device : 2vp4ff672-5

Number of Slices:	79	out of	3008	2%
Number of Slice Flip Flops:	66	out of	6016	1%
Number of 4 input LUTs:	151	out of	6016	2%
Number of IOs:	25			
Number of bonded IOBs:	25	out of	348	7%
Number of GCLKs:	1	out of	16	6%

Timing Summary:

Speed Grade: -5

Minimum period: 5.316ns (Maximum Frequency: 188.103MHz)

Minimum input arrival time before clock: 3.456ns

Maximum output required time after clock: 4.061ns

Device utilization summary for traffic light controller :

Selected Device : 2vp4ff672-5

Number of Slices:	20	out of	3008	0%
Number of Slice Flip Flops:	28	out of	6016	0%
Number of 4 input LUTs:	40	out of	6016	0%
Number of IOs:	18			
Number of bonded IOBs:	18	out of	348	5%
IOB Flip Flops:	12			
Number of BRAMs:	1	out of	28	3%
Number of GCLKs:	1	out of	16	6%

Timing Summary:

Speed Grade: -5

Minimum period: 3.872ns (Maximum Frequency: 258.231MHz)

Minimum input arrival time before clock: 3.660ns

Maximum output required time after clock: 4.214ns

Device utilization summary pre computation logic based traffic light controller:

Selected Device : 2vp4ff672-5

Number of Slices:	18	out of	3008	0%
Number of Slice Flip Flops:	23	out of	6016	0%
Number of 4 input LUTs:	37	out of	6016	0%
Number of IOs:	14			
Number of bonded IOBs:	14	out of	348	4%
IOB Flip Flops:	8			
Number of BRAMs:	1	out of	28	3%
Number of GCLKs:	1	out of	16	6%

Timing Summary:

Speed Grade: -5

Minimum period: 3.984ns (Maximum Frequency: 250.995MHz)

Minimum input arrival time before clock: 4.193ns

Maximum output required time after clock: 4.214ns